Technical Report

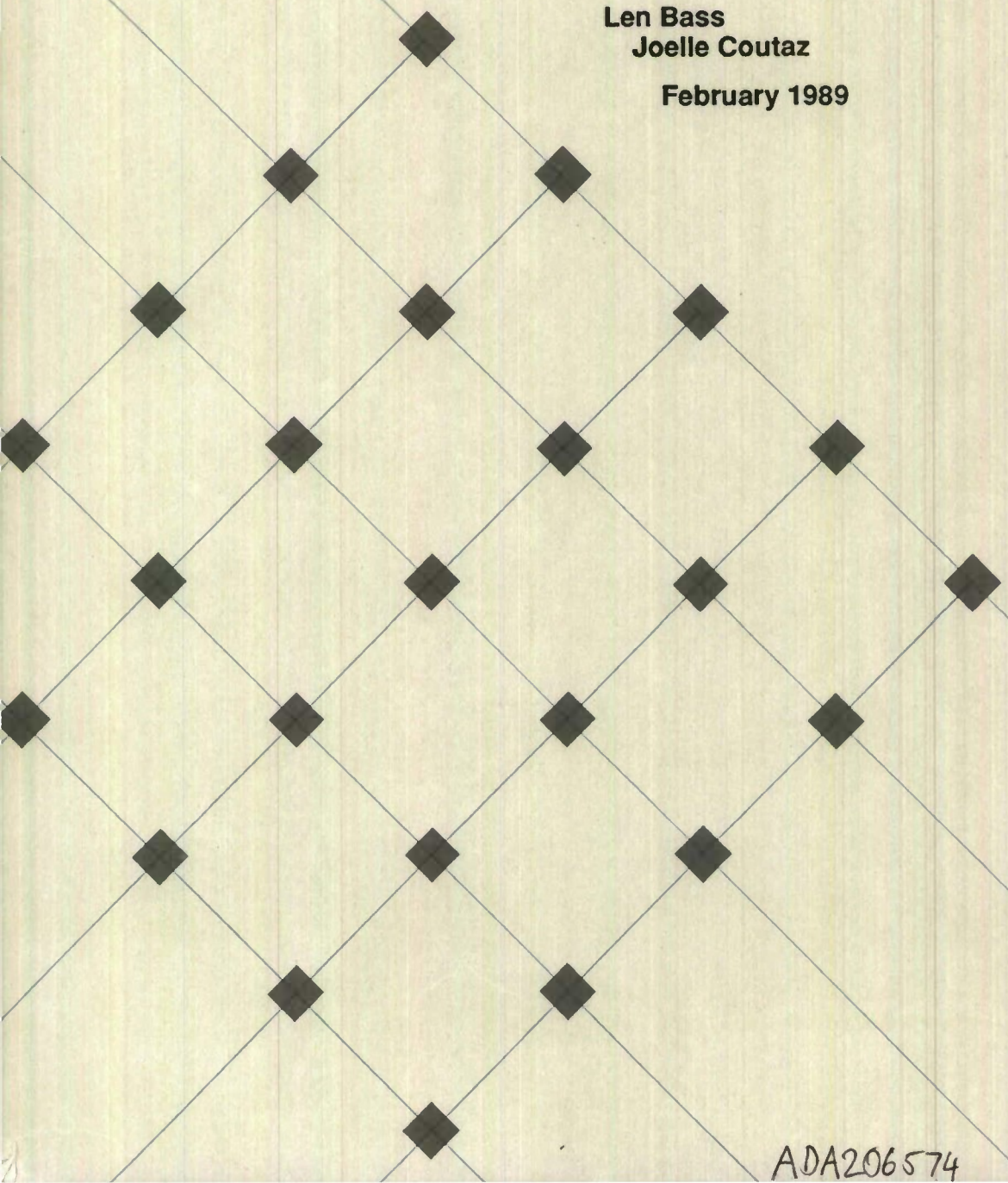
CMU/SEI-89-TR-4
ESD-TR-89-004

Carnegie-Mellon University
Software Engineering Institute

Human-Machine Interaction Considerations for Interactive Software

Len Bass
Joelle Coutaz

February 1989



Technical Report
CMU/SEI-89-TR-4
ESD-TR-89- 004
February 1989

Human-Machine Interaction Considerations for Interactive Software

Len Bass

User Interface Prototyping Project

Joelle Coutaz

Laboratoire de Genie Informatique (IMAG)
Grenoble, France

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

A previous version of this report appeared as follows: Coutaz, J., & Bass, L., *Ergonomics and Software Principles for the Construction of Interactive Software*, Research Report 731-I, Laboratoire de Genie Informatique, Grenoble, France.

Table of Contents

Human-Machine Interaction Considerations for Interactive Software

Abstract: This document introduces current concepts and techniques relevant to the design and implementation of user interfaces. A user interface refers to those aspects of a system that the user refers to, perceives, knows and understands. A user interface is implemented by code that mediates between a user and a system. This document covers both aspects.

1. Introduction

This document introduces current concepts and techniques relevant to the design and implementation of user interfaces. A user interface refers to those aspects of a system that the user refers to, perceives, knows and understands. A user interface is implemented by code that mediates between a user and a system. This document covers both aspects.

The first chapter is an introduction to the psychology of human-computer interaction. It presents the theoretical models that have had a significant impact on the evolution of the field. These models offer a way to organize the design process and help understand the cognitive processes involved in interacting with a computer.

The rest of the document is concerned with the software design of user interfaces and shows how the principles established by the cognitive principles can be put into practice. Following a presentation on the abstractions involved in the organization of an interactive system, attention is then directed to the tools for constructing user interfaces: windowing systems, toolkits and user interface management systems.

2. Models and Design Guidelines

Human-computer interaction is extensively cognitive. Even the most routine of activities, such as text editing, involves problem solving, requires the formulation of sequence of commands and implies the communication of these commands to the computer. To match the user's tasks, designers must go beyond their intuitive judgments and exploit ideas from cognitive psychology and human factors. These ideas may be classified into three categories:

- Theoretical models
- Practical guidelines
- Test strategies

The tutorial concentrates on some of the significant theories such as the Model of Human Processor [Card 83], GOMS [Card 83], the theory of Action [Norman 86] and the theory of Knowledge [Shneiderman 87]; it also briefly presents some practical guidelines based on these theories, on the Command Language Grammar [Moran 81] in particular. [Shneiderman 87] can be consulted for detailed comments on test strategies.

2.1. Models from Cognitive Psychology

2.1.1. Overview of the Human Processor Model

The Human Processor Model represents an individual as an information processing system. This system is comprised of three interdependent subsystems and operates according to a set of principles. As Figure 2.1 shows, the subsystems include perceptual, motor and cognitive systems. Each one is comprised of a processor and a memory. Processors and memories are characterized by parameters:

- τ, the processor cycle.
- m, the storage capacity in items.
- d, the decay time of an item, the time after which the probability of retrieving the item is less than 50%.
- k, the type of item held in memory (e.g., symbolic, physical).

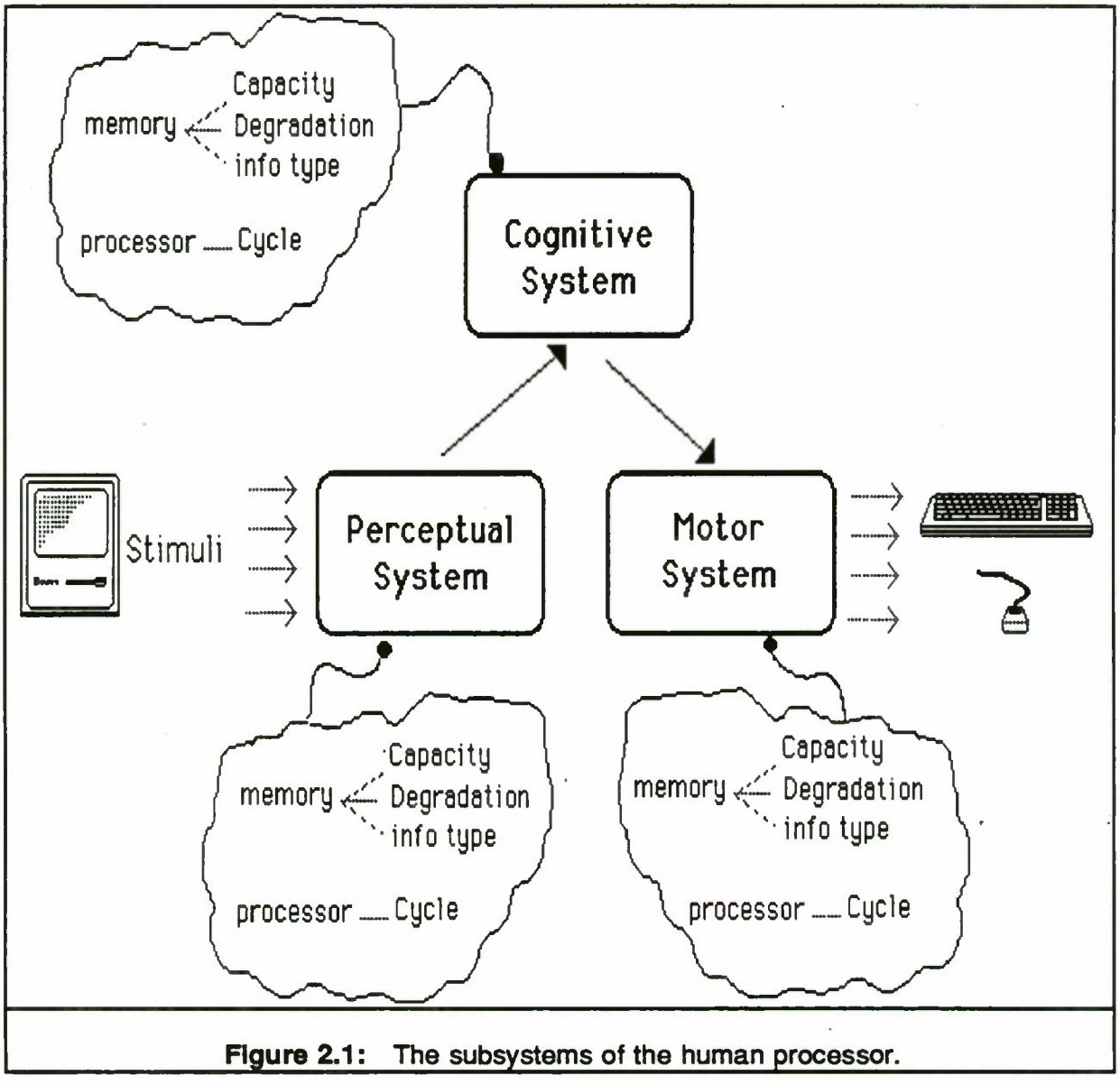


Figure 2.1: The subsystems of the human processor.

The general principles of operations that Card, Moran and Newell proposed include:

- The Encoding Specificity Principle: "Specific encoding operations performed on what is perceived determine what is stored, and what is stored determines what retrieval clues are effective in providing access to what is stored." [Card 83, p. 27]

- The Discrimination Principle: "The difficulty of memory retrieval is determined by the candidates that exist in the memory relative to the retrieval clues." [Card 83, p. 27]

- The Rationality Principle: "A person acts so as to attain his goal through rational action, given the structure of the task and his inputs of information and bounded by limitations on his knowledge and processing ability." [Card 83, p. 27]

- The Problem Space Principle: "The rational activity in which people engage to solve a problem can be described in terms of (1) a set of states of knowledge, (2) operators for changing one state into another, (3) constraints on applying operators, and (4) control knowledge for deciding which operator to apply next." [Card 83, p. 27].

- The last two principles have served as a basis for the model presented in Section 2.2.

The following subsections describe the usefulness of the model from the point of view of the computer scientist.

2.1.2 The Perceptual System

The perceptual system consists of a set of subsystems, each one specialized in the processing of a particular class of stimuli. A stimulus is a physical phenomenon that can be detected by a perceptual subsystem. A perceptual subsystem includes a processor, sensors and memory buffers called the visual image store (for the visual subsystem) and the auditory image store (for the auditory subsystem).

The visual image store holds the output of the visual sensory subsystem. It contains the physical representation of some stimuli, i.e., a coding that characterizes the physical properties of the stimuli. For example, in the visual image store represented in Figure 2.2, the coding of the character P expresses some shape and size but does not express the recognition of the character. Recognition is performed by the cognitive system described in Section 2.1.4.

A stimulus which impinges upon the retina at time τ, is available in the visual store at time $\tau+\tau_S$, where τ_S is the cycle of the visual processor. The mean cycle of the visual processor is around 100 msec and varies with the intensity of the stimuli. This means that an individual generally needs 100 msec before having the feeling of perceiving. In other words, two images produced in the same cycle are perceived as a single one. This result means that refreshing the screen will appear instantaneous to the user if the image can be produced in less than 100 msec. Satisfying the 100 msec constraint relies heavily on hardware technology and has impact in software construction. An example is the work of Uebbing [Uebbing 86] in analyzing the objects in object-oriented languages. One drawback of object-oriented languages is the overhead due to message passing. Uebbing comments on an interesting experiment about code optimization. He shows how to reorganize objects and minimize message passing times. Knowing :

1. τ_m, the transfer time of a message between two objects (e.g. 0,04 msec for Objective-C on a MC68010).

2. n, the number of elementary objects comprised in a compound object.

then, the total time τ spent in message passing to redraw the compound object is $\tau = nt_m$. If τ is greater than the threshold which is a function of the visual processor cycle τ_S, then it is desirable to:

- Minimize message passing by reorganizing the compound object.

- Draw part or all of the compound object with low-level tools (even assembly language if this turns out to be necessary).

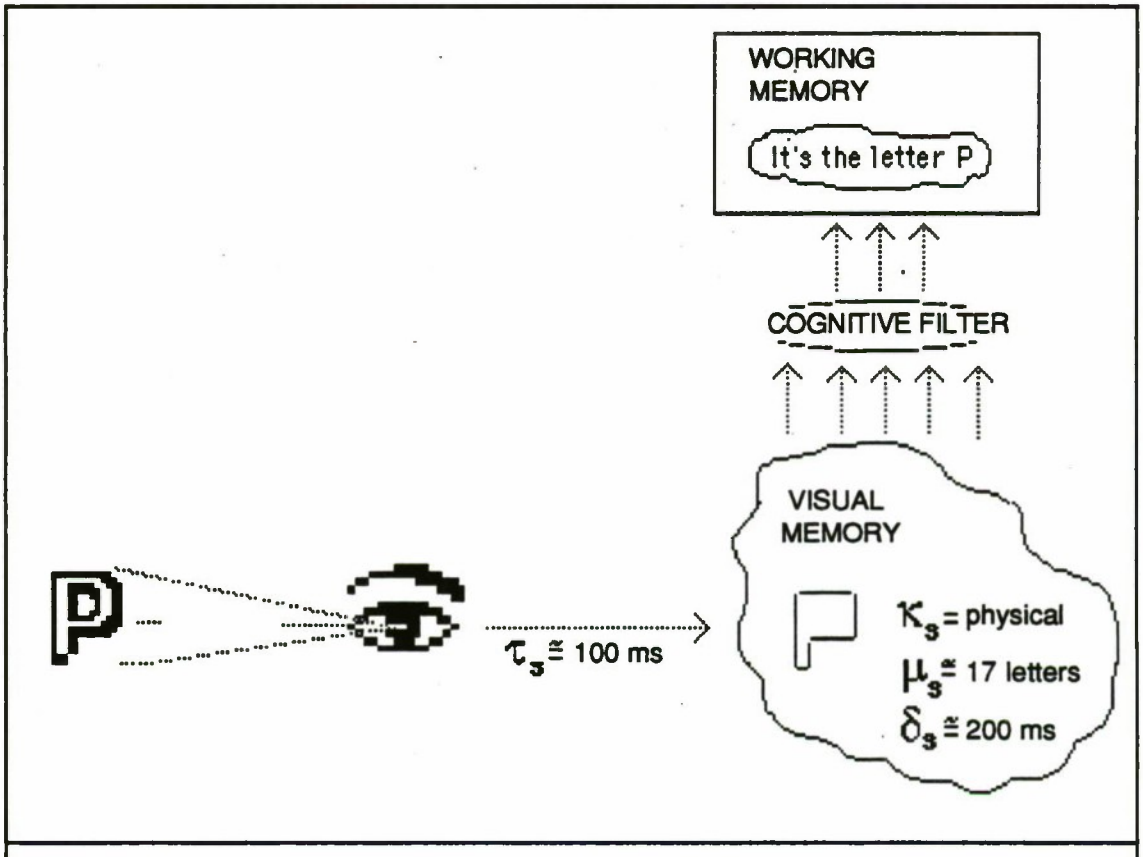


Figure 2.2: The visual sensory subsystem and its relationship with the cognitive system.

message passing is the notion of windowing service through local area networks. This technique will be subsequently developed in Sections 3 and 4. X-Windows [Scheifler 86], which is such a server, is able to handle mouse events fast enough to make immediate feedback possible without making the user aware of the network.

2.1.3. The Motor System

Shortly after information has reached a perceptual memory, the cognitive system receives symbolically coded information in its working memory. The cognitive system uses previously stored information in the long -erm memory to make decisions about how to respond: the model views thought as translated into actions by activating muscle movements. The Motor System is responsible for movements. Movements that are of interest for human-computer interaction include arm-hand and eye-head gestures.

A movement is made of a sequence of discrete micromovements. Each micromovement requires one cycle τ_m of the motor system. The mean value for τ_m has been evaluated to 70 msec. With the hypothesis that a movement results from a

sequence of micromovements, it is possible to compute the theoretical time to move the hand to a given target. Figure 2.3 shows the initial situation: the hand is located in X_0, at a distance D from the target (X_0 = D). The size of the target is S. After the first micromovement, the hand is in X_1, then in X_2, etc. One can show that the time T required to place the hand on a target depends on the required relative precision, that is on the ratio between the distance and the size of the target:

$$T = I \log 2(D/S + 0.5)$$

where I is a constant determined experimentally (around 100 msec). This equation is known as Fitts's law.

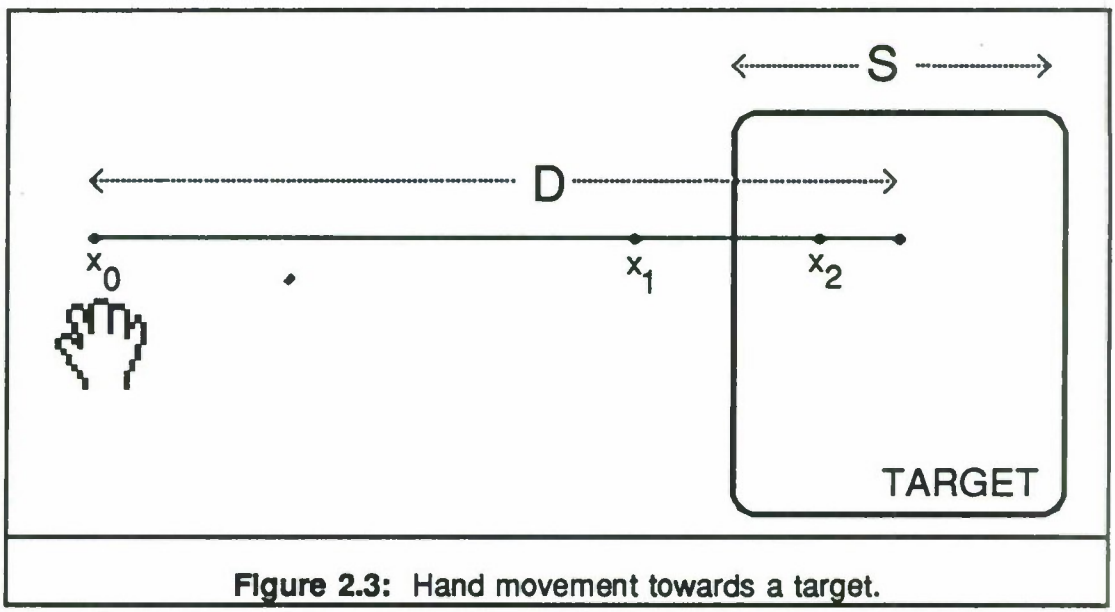


Figure 2.3: Hand movement towards a target.

Fitts's law can be usefully applied to determine the time spent in hand homing between input devices or in object selection on the screen. Such computations can serve as a quantitative evaluation of alternatives between syntaxes.

2.1.4. The Cognitive System

There are two important memories in the cognitive system: the working memory and the long-term memory (see Figure 2.4). The working memory (also called short-term memory) holds information under current consideration just like the general registers of a computer. It contains the intermediate product of thinking, the representations produced by the perceptual system, and a subset of activated items extracted from the long-term memory. The long-term memory stores knowledge for future use in the form of symbols, called chunks.

A chunk is a cognitive unit whose nature depends on the user. For example, SNCF is made of four chunks (i.e. the four letters S, N, C and F) for someone who does not know that SNCF is the acronym for the French train company, whereas it is a single chunk for French people. Chunks can be organized into larger units and be related to other chunks. For example, the chunk "car" is composed of the chunks "wheel," "body,"

etc., and the chunk "weather" is related to the chunks "sun," "rain," "cloud." Semantic networks have been widely used to represent such relationships between pieces of knowledge.

When a chunk is activated, previously activated chunks are less available because of limited capacity of the working memory. The new chunks interfere with the other ones which tend to disappear from the working memory if they are not reactivated. Note that the working memory behaves like the working set of virtual memory paging systems: when a page fault occurs (i.e., when a chunk is activated), pages in the main memory that have not been used (i.e., chunks that have not been reactivated) are swapped out to let the last referenced page be installed in the main memory. The Room model presented in Section 4.7 illustrates this notion of "cognitive working sets" by organizing the task space of the user in closely related windows.

The capacity of the long-term memory is infinite: there is no erasure from the long-term memory, but retrieval of a chunk may fail. This failure may have several causes: no association can be found or similar association to several chunks interfere with the retrieval of the target chunk. As a consequence, the best way to remember something later and avoid chunk interference is to associate it with chunks of the long-term memory in a unique way.

While the capacity of the long-term memory is infinite, that of the working memory is very limited. It has been demonstrated that the capacity of the short-term memory is 5 ± 2 [Miller 75]. As a result, not only should software engineers pay attention to short-term memory overload but also should devise effective electronic extensions. Section 2.6 shows that menus and forms constitute such appropriate extensions.

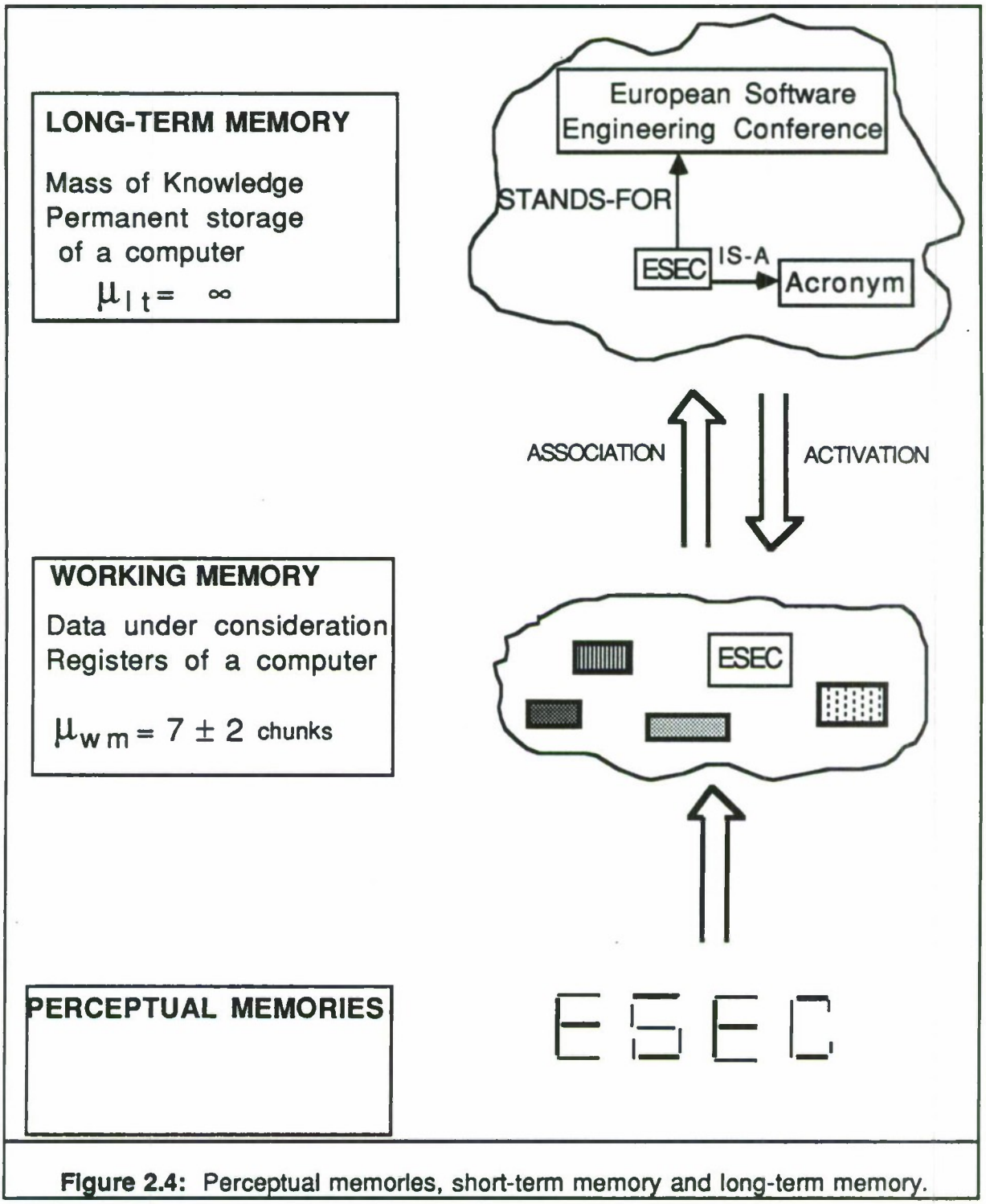


Figure 2.4: Perceptual memories, short-term memory and long-term memory.

2.1.5. Evaluation of the Human Processor Model

Clearly, the Human Processor model is a simplification of the complex state of present knowledge in cognitive psychology. However, it provides the computer scientist with a comprehensible framework on which various aspects of this knowledge can be

gradually plugged. Actually, the goal of Card et al. goes beyond providing a framework for thoughts. The goal is to create a new discipline that would combine characteristics from fundamental and applied sciences. As in physics, this discipline would allow the designer to perform approximate evaluations. With the help of a technical theory [Newell 86], it would be possible to elaborate models that would allow the designer to answer questions about a particular phenomenon in human-computer interaction. The model of the Human Processor is a step towards this technical theory. For doing so, it introduces parameters that help in formalizing user performance and making predictive evaluations.

Unfortunately, the parameters of the model of the Human Processor are useful for computing low-level behavior only. They are useful in determining the optimal rate for refreshing the screen; they stress the incidence of size targets on the effectiveness of selection actions; they explain why special attention should be devoted to short-term memory overload. Although mathematical expressions bring some scientific coloration to the development of a domain, the parameters of the model of the Human Processor are driven purely by performance considerations. They do not help in the understanding of the underlying cognitive processes that lead to such performance. The principles of operation that accompany the model are an attempt in this direction. One of them, the principle of rationality, serves as a basis to goals, operators, methods, and selection [Card 83], described in the next subsection.

2.2. Practical Guidelines for Design

GOMS:

- Is based on the theoretical hypothesis described in the previous subsection: a human being acts in a rational manner.
- Is a model for the performance of the user who does not make errors.
- Structures the cognitive activity involved in accomplishing a task into four components: Goal, Operators, Methods, Selection.

A goal is a symbolic structure that:

- Defines a desired state.
- Determines the set of methods which lead to this goal.
- Constitutes a backtrack point in case of failure.

Goals are organized hierarchically. The leaves of the hierarchy are operators. For example, when starting to edit a document, the user has the top level goal "edit-manuscript." The user segments this larger task into smaller tasks and devises the subgoals to achieve the subtasks. Figure 2.5 gives an example of such a subtask, which consists of transposing two words.

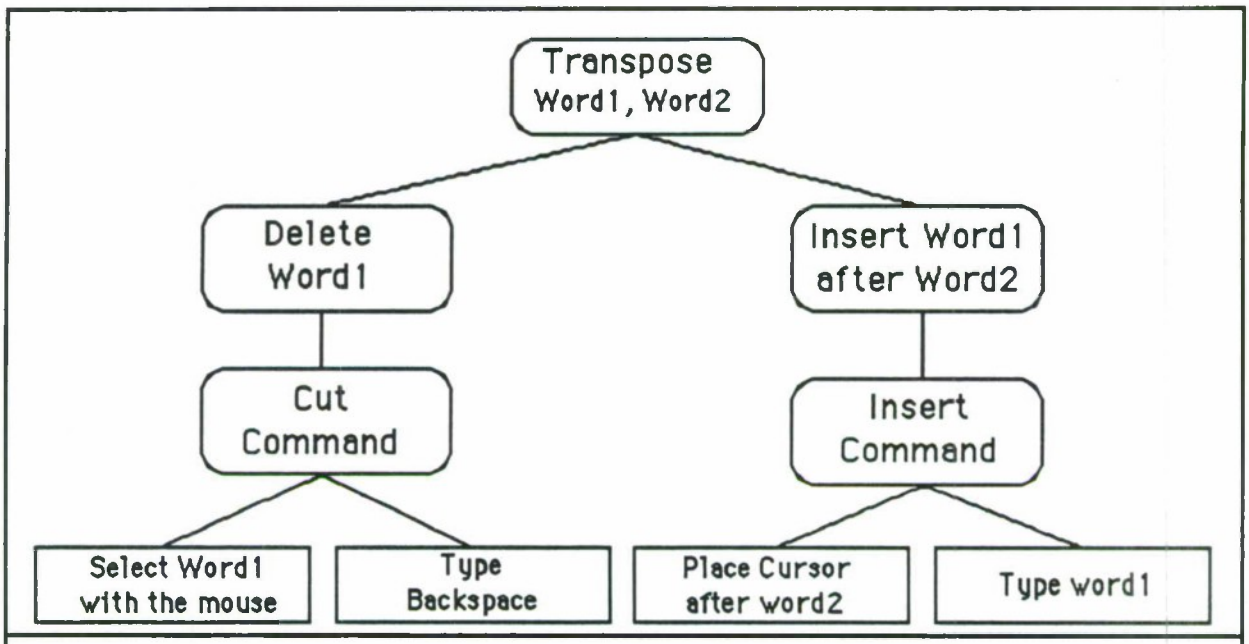


Figure 2.5: The decomposition of a goal (transpose two words) into a hierarchy of subgoals. The leaves of the tree denote physical actions. The illustration does not make explicit the selection rules applied by the user when several methods lead to the same goal.

An operator:

- Is a perceptual, a motor or a cognitive action.
- Provokes a change in the mental and environmental state.
- Is characterized by I/O parameters and an execution time.

A method:

- Describes the know-how. The know-how is made of learned procedures that the user already has at execution time. They are not plans created at execution time. The learned procedures express skill built from prior experience. They reflect the knowledge of the exact sequence of steps to accomplish a task
- Is a sequence of conditions about goals and operators.

A selection rule determines the choice between the methods that achieve the same goal.

GOMS can be used to model and predict the user's behavior at various levels of abstractions. One application of GOMS at a low level of abstraction is the KEYSTROKE level model [Card 83] which, given a command language, allows the designer to predict the time needed by the user to enter a command.

To summarize, GOMS:

- Is useful for predicting errorless behavior.
- Does not deal with concurrent operations: the behavior is assumed to be linear. The goal stack model does not fit non linear planning; and non linear planning is required to deal with the user's interruptions (e.g. errors).
- Is behaviorist: it is a model about performance. It is not cognitive, as is the theory of action in the next section.

2.3. The Theory of Action and Conceptual Models

One of the goals of cognitive engineers is to identify and understand the principles that guide the actions of the individual. The theory of D. Norman relies on the hypothesis that the user elaborates conceptual models and that task accomplishment involves several stages [Norman 86].

2.3.1. Conceptual Models

A conceptual model:

- Is a mental representation of oneself and of the environment.
- Depends on previous knowledge and understanding.
- Is modified by the nature of the interaction.

When considering the interaction of a user with an artifact, it is important to consider two conceptual models (the designer's and the user's conceptual models) and the notion of system image. If the artifact is a computer, there is also the system's model to consider. Figure 2.6 represents these models.

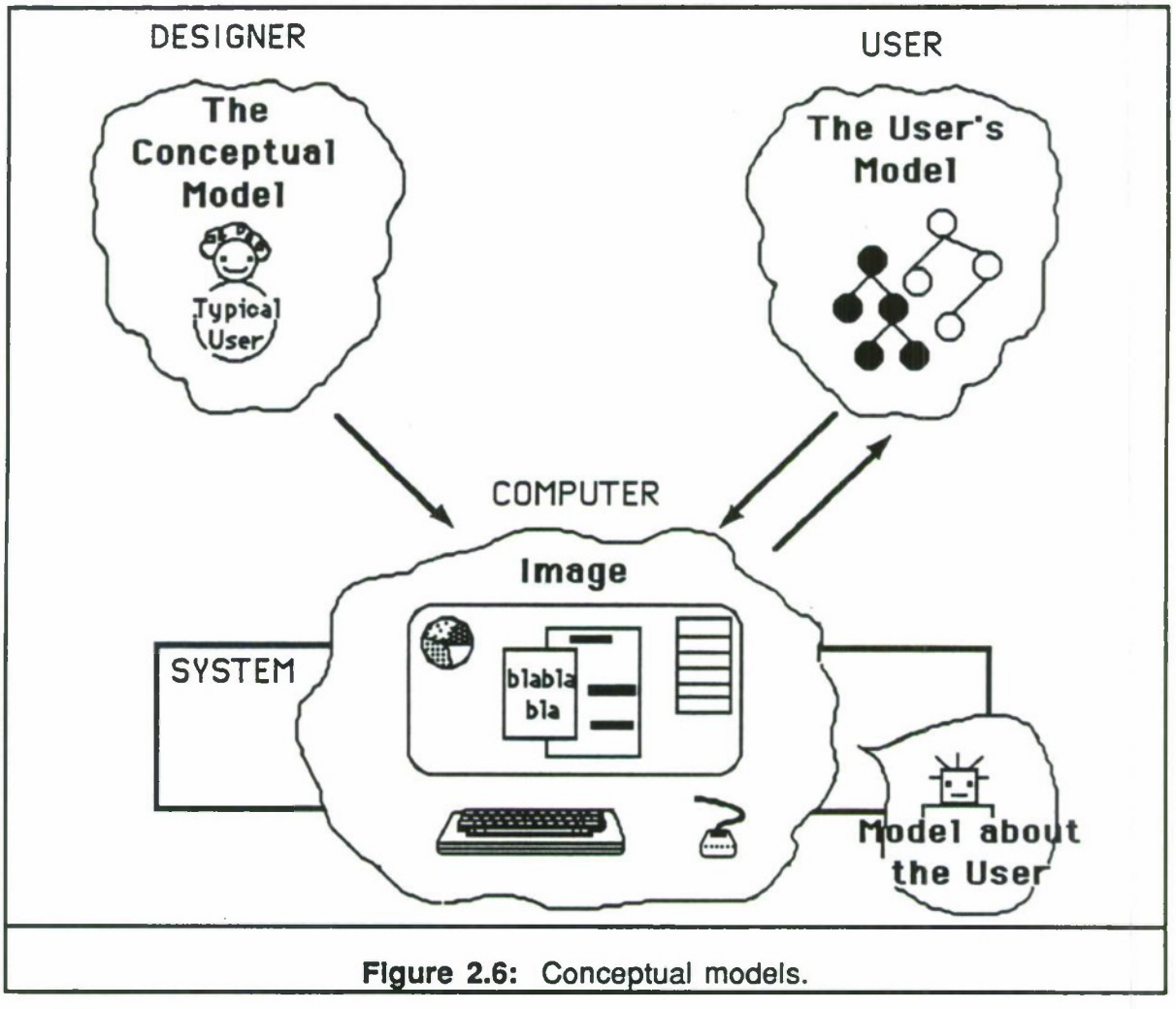


Figure 2.6: Conceptual models.

The Designer's Conceptual Model:

- Is the model that the designer devises for the artifact.
- Relies on the representation that the designer has about the typical user of the artifact. Ideally, this conceptualization is based on a thorough analysis of the user's tasks, requirements, capabilities, background and experience.

The User's Conceptual Model:

- Results from the user's interpretation of the system image.
- Defines the "view" that the user has about the system.

The System Image:

- Results from the physical structure that has been built (artifact).

- Should be explicit, intelligible and consistent, so that the user may elaborate a conceptual model compatible with the design model. The burden is placed on the image that the system projects. Accomplishing a task will be easier or harder, depending on the system image.

The System's Model:

- Is the model that an intelligent program might build about the user.
- Allows for automatic customization.

2.3.2. Toward a Theory of Action: Stages of User Activities

Accomplishing a task involves approximately seven stages (see Figure 2.7):

1. *Establishing the Goal*

   A goal is a mental representation of the desired state. It is expressed in terms of psychological variables. The system state is defined by the value of its physical variables, such as the location of the cursor or a sequence of words that forms a sentence. The user compares the system state to the goal. To do so, the system state is translated into a psychological representation.

2. *Forming the Intention*

   The evaluation of the distance between the goal and the translated state of the system gives rise to an intention. An intention is the decision to act toward achieving a goal. An intention is stated in psychological terms. It specifies the meaning of the input expression that is to satisfy the user's goal. To do so, the user must know the mapping between the psychological variables and the physical variables; for example, the user must have established the correspondence between the notion of insertion point, which is a psychological variable, and the location of the cursor, which is a physical variable; As another example, in order to achieve the goal "delete word1" in Figure 2.5, the user must know the link between suppressing a word, which is a psychological notion, and the command "cut," which is a physical input expression. The user must know the effect, the meaning, of the command "cut."

3. *Specifying the Action Sequence*

   The intention must be translated into a sequence of actions. To do so, the user has to know the mapping between the physical variables and the physical control mechanisms. A physical control mechanism allows for the modification of physical variables. The specification of an action sequence is a mental representation of the actions to perform on the physical control mechanisms. It prescribes the form of the input expression that has the desired meaning. For example, the user must know that the location of the cursor can be modified with the mouse. In the example in Figure 2.5, the user knows the syntactic-lexical definition of the command "cut."

4. *Executing the Action*

   The execution of an action is the manipulation of physical control mechanisms.

5. *Perceiving the System State*

   The system state is embedded in an output expression. The perception of this expression is the translation of the physical variables into psychological variables. For example, after typing the character backspace in Figure 2.5, the user perceives that the output expression no longer contains the word displayed in reverse video in the previous output expression.

6. *Interpreting the System State*

   The interpretation of the output expression results in determining the meaning of the output expression. For the example in Figure 2.5, the disappearance of the word is interpreted as the deletion of the word.

7. *Evaluating the System State with Respect to the Goals*

   The evaluation establishes the relationship between the meaning of the output expression and the user's mental goal. This evaluation may result in a modification or in continuing to the next step in the plan.

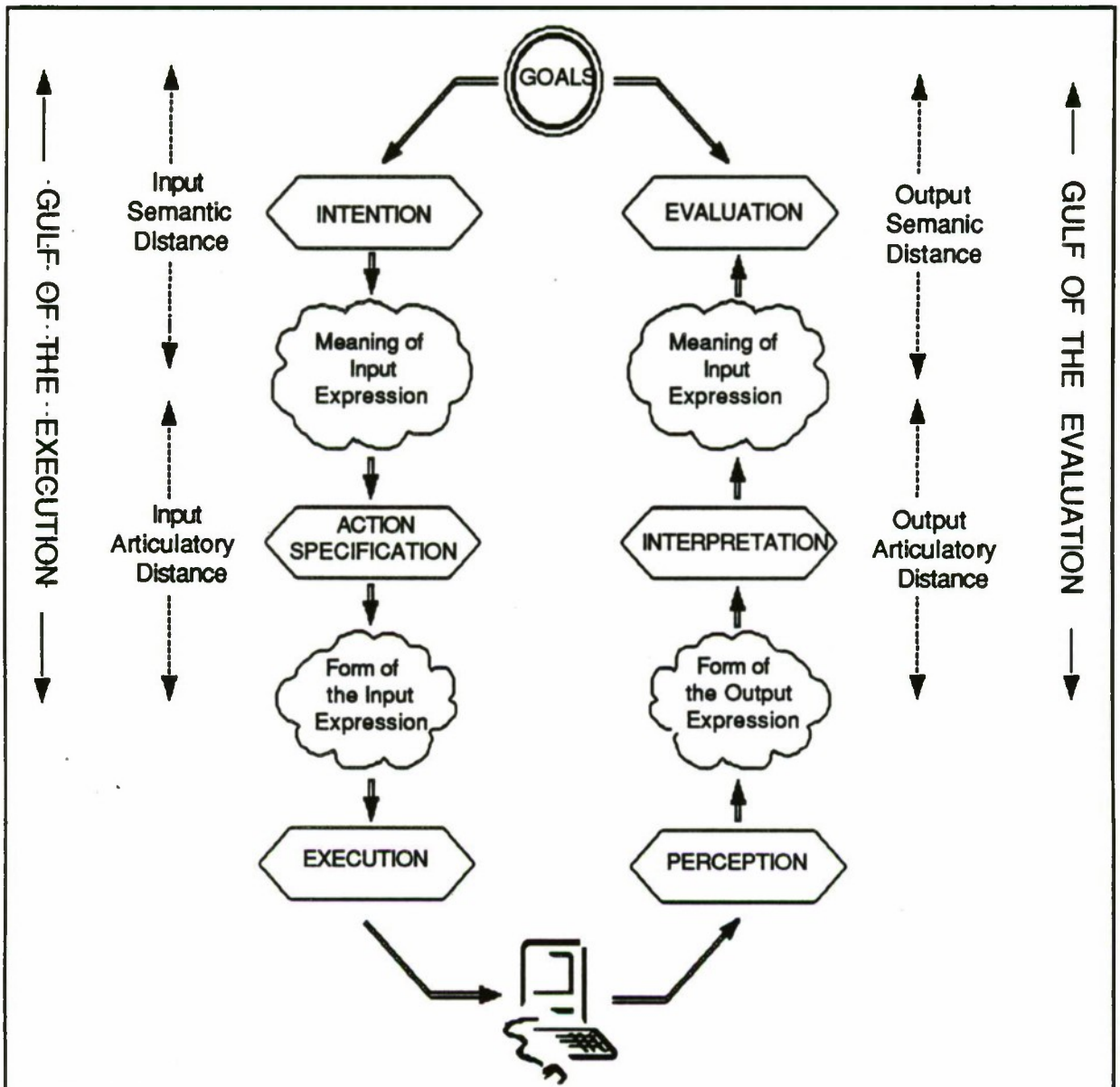


Figure 2.7: Stages in user's activities and gulfs of execution and evaluation.

Accomplishing a task:

- Does not necessarily require the presence of the seven stages.
- Does not require these stages to happen in a specific order.
- Creates different needs at different stages. For example, menus can assist in the stage of forming an intention and specifying an action, but frequently make execution more clumsy.
- Does require a translation between the psychological representations and the physical presentations. This translation reveals the existence of

a gap between the mental world and the physical world. Norman calls this discrepancy a "gulf."

The gulf between the user and the system is two-way: from the mental representation to the physical presentation, and from the physical world to the mental world. The first gap is called the gulf of execution, whereas the second is the gulf of evaluation.

The gulf of execution consists of the semantic distance and the articulatory distance. The semantic distance is covered by the intention, which goes from the goal to the specification of the meaning of an input expression that is to satisfy the goal. The articulatory distance is covered by the action specification, which goes from the meaning of the input expression to its syntactic/lexical form.

The gulf of evaluation also consists of an articulatory distance and a semantic distance, covered respectively by the interpretation of the output expression and the evaluation of the meaning of the output expression.

In summary, this theory stresses the fact that the accomplishing of a task involves several stages, that each stage has its own possibly conflicting needs, that these needs result from the gulf between the mental representation and the physical presentation, and that this gulf should be bridged by the system designer as much as possible through the system image. Conversely, if the matches between the psychological and the physical variables are weak, the user has to bridge the gulfs by creating more plans, more action sequences and more interpretations that move the psychological description closer to the physical requirements.

Opposite GOMS, which provides the designer with a synthetic view of human behavior, Norman's theory of action analyzes the mental processes that lead to such behavior. Whereas GOMS is limited to the ideal case of errorless interaction, Norman stresses the difficulties encountered by the user and provides the designer with a general framework for explaining the cause of errors. GOMS is a quantitative model about human performance, whereas Norman's theory of action is an informal, explanatory, cognitive model about human behavior. The informal nature of Norman's theory prevents the designer from making predictive evaluations. However, such a theory can serve as a basis for the development of evaluation techniques (e.g., ETIT [Moran 83]). The intuitive view of Norman's theory is interestingly complemented by ACT* [Anderson 83], a formal theory of human cognition based on production systems.

2.4. Theory of Knowledge: The Semantic/Syntactic Model of Knowledge

The nature of knowledge has been studied extensively, resulting in various theories about how knowledge is organized and exploited. This section, first describes briefly a general theory of knowledge, as well as the semantic/syntactic model of knowledge, useful in the context of user interface design.

2.4.1 A General Theory [Simon 84, Card 83]

Subsection 2.1.4 explains that knowledge is organized as a network of chunks. This network contains two classes of information:

- Factual knowledge: a set of assertions, predicates, known facts with possibly confidence factors

- Procedural knowledge: a set of procedures that describe the know-how. A procedure is an elementary action, such as a computer instruction. Unlike the computer instruction set, the procedure set of the cognitive processor evolves with time.

In the context of human-machine interaction, the chunks of interest here are those that constitute the user's conceptual model. This conceptual model contains facts and know-how about the system. Today, it is widely agreed that these facts and skill can be classified into two categories: syntactic knowledge and semantic knowledge (see Figure 2.8) [Shneiderman 87].

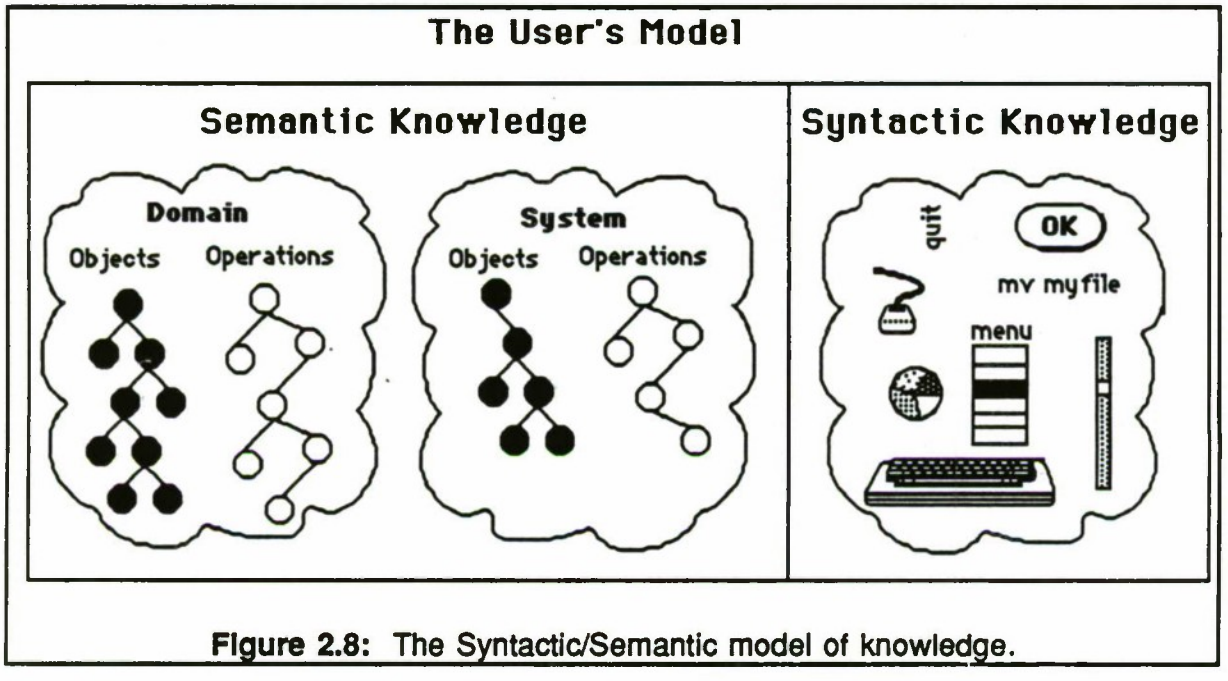


Figure 2.8: The Syntactic/Semantic model of knowledge.

2.4.2. Syntactic/Semantic Knowledge

Syntactic knowledge:

- Represents the lingulstic conventions that the user must know to specify requests to the system (input expressions) or to interpret responses from the system (output expressions). These conventions allow the user to communicate with the system image.

- Is system dependent.

- Is arbitrary, inconsistent, difficult to retrieve and has many other negative qualities.

- Must be acquired by rote memorization and repetition.

Semantic knowledge is:

- An organized hierarchy of factual and procedural concepts: factual concepts are in the form of objects or data. Procedural concepts are operations on objects or procedures on data. In addition, a distinction should be made between domain-dependent objects and operations, and system-dependent objects and operations.
- Potentially transferable across different computer systems.
- Independent of syntactic details.
- Acquired by meaningful learning.

The distinction between syntax and semantics, and between domain-dependent concepts and system-dependent concepts, match the usual forms of competence: a user may be incompetent in a domain but skillful at using a particular computer. Conversely, the user may be knowledgeable in a field, but ignorant in the use of a particular computer system.

2.5 Theoretical Models: Summary

Models presented so far are concerned with phenomena related to human-computer interaction.

- Some models, such as the Human Processor Model, GOMS and Keystroke, are useful for making quantitative predictions about a particular design. However, by oversimplifying the real world, they are too limited in scope and too low level.
- Other models, such as Norman's Theory of Action and Shneiderman's model of Syntactic/Semantic Knowledge, provide the designer with explanations about the cognitive behavior of the user. Although they take a more realistic view of the real world, these models lack of a scientific formalism makes them unusable as predictive tools.

The user interface designer has the difficult task of integrating these various theories into a unique "easy-to-whatever" computer system! Combining all of these principles leads directly to some kind of combinatory explosion. Combinatory explosion may be avoided with the use of heuristics. Heuristics does not guarantee an optimal solution, but it provides a reasonable answer. The following section we introduces some general heuristics that needs to be flavored with the peculiarities of the specific case at hand.

2.6. Practical Guidelines: Methods and Golden Rules

The general method presented in this section is an application of the Command Language Grammar [Moran 81], although the Command Language Grammar (CLG) is not a methodology. CLG conveys a type of top-down approach that can be found useful as a framework for designing user interfaces. CLG is a grammatical structure to represent computer systems at various levels of abstractions. Each level of representation defines a particular view of the system, and each view results from an

analysis that any competent designer should perform. Figure 2.9 illustrates the principles of CLG, whose terminology is explained in Section 2.6.1.

2.6.1. General Method for User Interface Design

The design of a particular interactive system may be structured along five axes:

1. Definition of the profile of the user based on a general classification (notion of novice, expert, and occasional user, combined with the notion of semantic and syntactic knowledge).

2. Definition of the profile of the tasks: utility of the system according to the needs of the user. This constitutes the task level of CLG. It consists of defining the domain-dependent entities as perceived by the user:

   - the task entities of the domain.
   - the tasks to be performed in the domain.
   - the decomposition of the tasks into a hierarchy of subtasks.
   - the task procedures (methods) to perform the various tasks.
   - the privileged tasks, i.e., tasks that need special attention due, perhaps, to their frequency.

3. Definition of system-dependent notions to implement the domain-dependent concepts. This constitutes the semantic level of CLG. It includes:

   - the conceptual entities, which act as the electronic representations of the conceptual objects and of the additional entities that the system uses for its own purposes.
   - the user and system conceptual operations to manipulate the conceptual entities (looking for an information on the screen is considered a user conceptual operation).
   - the semantic procedures (methods) expressed in terms of the user, and system conceptual operations to perform the tasks defined in the task level.

4. The definition of the structure of the dialogue in layers of increasing complexity and leading to task closure. This is the syntactic level of CLG. It includes:

   - the commands and their arguments.
   - the clustering of commands into contexts and the mechanisms for switching between contexts.
   - the syntactic procedures (methods) expressed in terms of the commands, as well as in terms of the conceptual operations of

the user. A syntactic procedure shows how to perform a task defined at the task level.

5. Definition of the interaction style, choice of the lexical details (e.g. world metaphor vs. conversational metaphor). This is the interaction level of CLG. It includes:

   - the interaction elements and the primitive actions performed by the user and by the system (keystroke and mouse selection are examples of user actions; prompts and responses are system primitive actions).

   - the order in which the interaction elements must be specified by the user or produced by the system.

   - the interaction procedures (methods) expressed in terms of the primitive actions and in terms of the conceptual operations of the user. An interaction procedure shows how to perform a task defined at the task level.

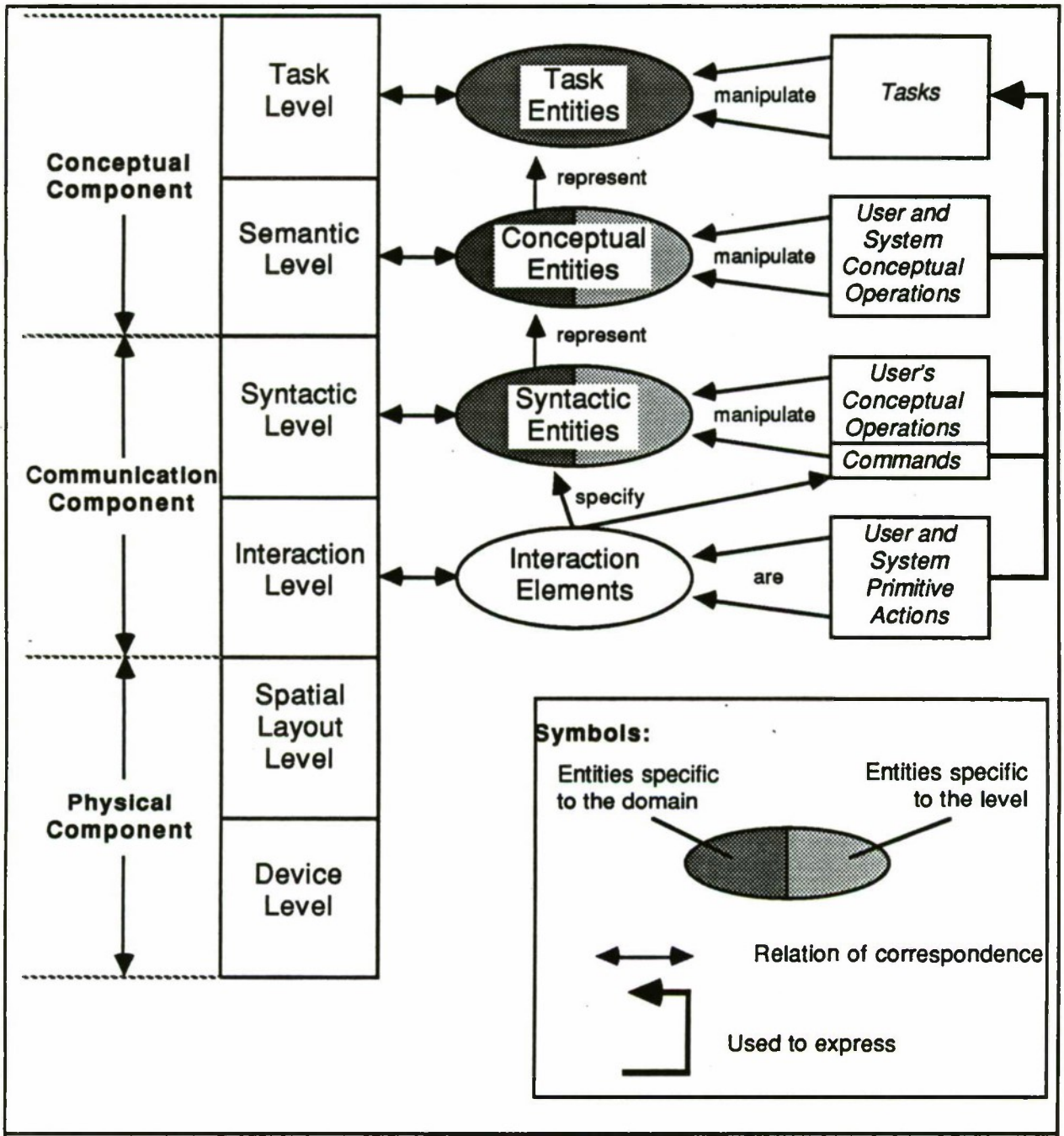


Figure 2.9: The CLG layers for the Design of User interface.

As the description of CLG shows, a particular system is fully described at various levels of abstraction. Each level manipulates its own entities and operators, but these elements are combined to fully describe the system. Each level can be viewed as a refinement of the previous one (i.e., higher in the hierarchy) and each level is independent of the following one (i.e., lower in the hierarchy). By following this hierarchical method, CLG yields a top-down approach to the design of a user interface.

Although CLG can be usefully exploited as a method, the basic difficulty for the designer is defining and structuring the user's tasks. If the description of the task domain does not match the mental representation and the cognitive processes of the user, the system will probably be hard to use and hard to learn. Unfortunately, an appropriate organization of the user's tasks requires an intensive knowledge in the domain of cognitive psychology, a knowledge that most computer scientists do not master.

The subsections that follow provide the designer with practical guidelines that may be useful to define the syntactic and interaction levels of CLG.

2.6.2. Guidelines

The guidelines presented in this subsection form a very small fraction of hundreds of rules currently available in the literature. For a more complete enumeration, refer to [Scapin 87, Shneiderman 87]. The guidelines that follow are a selection of general human factor principles that computer scientists may apply easily. They are organized as a set of seven guidelines: consistency, concision, cognitive load reduction, user-driven interaction, flexibility, dialogue structuring, and error prediction.

2.6.2.1. Guideline 1: Consistency

Consistency implies the absence of exception. Exceptions increase learning time and the likelihood of error. System consistency is a concern at all of the stages that D. Norman identified for modeling human-computer interaction. This subsection is limited to the stage of action specification and to the execution stage. Rules for the perception and the evaluation stages derive directly from those considered here.

- Consistency and the Action Specification Stage

  If a goal is similar in different environments, then the sequence of actions to accomplish the goal should be the same.

  For example, a user needs to "duplicate an object and print the copy of the object". The object may be a document or an electronic mail message. In both environments, the mail system and the document preparation system, the sequence of actions should be the same.

- Consistency and the Execution Stage

  The execution stage includes syntactic, lexical and pragmatic issues.

  - With regard to syntax, the designer should determine the order of command arguments. Experiments indicate that when commands share arguments, these arguments should appear in the same order in every command.

  - Note that the order does not always match the sequencing of natural languages and that there is a choice between postfixed notation and prefixed notation. It seems that for graphical environments, a postfixed notation is more appropriate whereas the prefixed notation is adequate for text-based interaction.

- With regard to lexical issues, naming should be consistent . If some function appears in different contexts, it should be designated with the same name.

  A counter example of this rule is the function "terminate" in the Unix world: to terminate a message in the mail system, the user must enter a single character line (character "."); to terminate the mail system, the user must type "q" (or "x" depending on how the user wants to reenter the mail system); typing "logout" terminates a Unix session.

- With regard to pragmatics issues, consistency recommends that spatial layout of output information should be preserved.

  This principle of locality helps the user anticipate gesture on system outputs. In particular, menu items should always appear in the same order. The order must primarily depend on a logical sequencing defined by the task; if the task does not show any logical order, the frequency criteria should be applied; however, if the frequency criteria is not applicable, alphabetical order should be used. Similarly, locality rules have been defined for forms: at the top of the form, the user should find the fields that must be filled whereas optional items can be gathered at the bottom. Note that this guideline is consistent with Fitts's Law: it minimizes hand movements.

2.6.2.2. Guideline 2: Conciseness

Conciseness is the harmonious combination of brief and powerful expressions. In computer-human interaction, conciseness is achieved in the form of abbreviations, macrocommands, cut and paste facilities, undo and redo features, and default values.

This section illustrates the difficulty in applying these guidelines with the use of judgement by the designer. For example, conciseness is desirable for the experienced user but not for the novice user. It is important to identify the end users of a particular interface and tailor the interface to their characteristics.

- Conciseness and Abbreviations

  Abbreviations are useful shortcuts for experienced users. Shortcuts are mandatory. For example, menus are adequate as a technique for minimizing memory load, but they are clumsy when considering the action specification stage (a Keystroke level model can be used to support this assertion). However, in order to be understandable, abbreviations should be derivable from precise rules.

  Common rules for deriving abbreviations include:

  1. Special character (e.g., escape or control) followed by a letter (usually the initial of the command name). EMACS is a good example of the application of this rule.

2. Vowel deletion. For example, the command delete would be abbreviated as "dlt".

3. Maximum truncation which consists of suppressing characters from command names as long as there is no ambiguity. For example, given the set of command names "compile, copy, delete," the rule respectively derives "com, cop, del".

4. Two character truncation. This rule applied to the set "compile, copy, delete," would derive "cm, cp, dl".

Figure 2.10 illustrates the results of a study that compares user performance according to the abbreviation rule [John87]. The response time is the mean time the user needs to enter an abbreviated command.

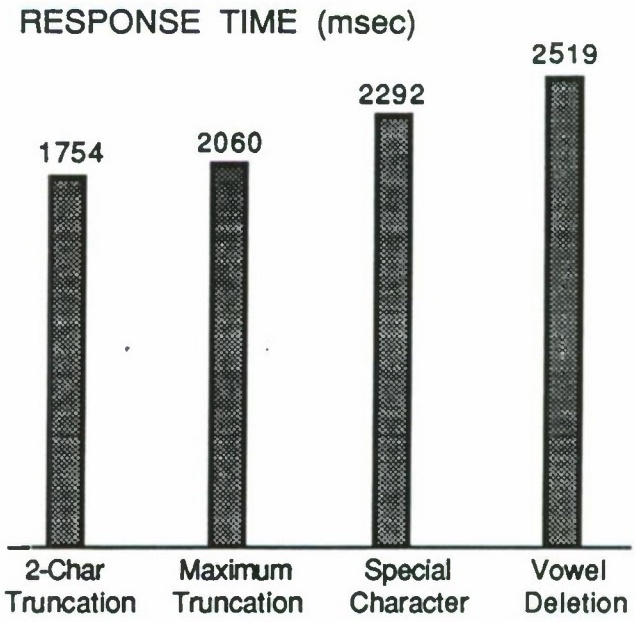


Figure 2:10: Comparative user performance according to the abbreviation rule.

- Conciseness and Macrocommands

  A macrocommand is to interaction languages what a procedure is to programming languages. It is an abstraction mechanism and an extension technique. As an abstraction mechanism, it matches human learning cognitive processes that encapsulate related pieces of knowledge into a "bigger" chunk. As an extension technique, it allows for combining generality and particularity.

Considering human-computer interaction, particularity denotes user specific needs. Norman's theory of action identifies the semantic distance between the formation of the intention and the elaboration of a plan of commands. One way to shorten the gulf is to provide the user with a high-level language that directly expresses the most frequent problem-solving plans. The drawback of a highly tailored language is the difficulty to express unusual tasks.

The conflict between particularly and generality has been solved in Unix and Lisp-based systems by providing the user with a fairly low-level, general purpose language to build new commands. These commands may encapsulate frequently encountered actions into a single parametrizable chunk. Unfortunately, the user interface for defining such macros forms a highly disappointing cognitive barrier to the newcomer or to the unmotivated user.

- Conciseness and Cut and Paste Facilities

"Cut and Paste" is the electronic version of manual patchwork. As with manual patchwork, it offers a way to reuse information. For example, it avoids the need to retype information, or it allows the user to enter information already provided by the system. Cut and paste is also a means for overcoming lack of integration between tools. For example, the user can develop a text with a special purpose text editor, then draw a picture with a sophisticated interactive editor, and eventually paste the picture into the text document. In integrated environments, there would be no need for the user to explicitly use different tools. In any case, cut and paste operations must appear consistent to the user.

An example of inconsistency is a round-trip transfer of information between MacDraw and MacPaint. MacDraw manipulates graphical objects such as circles and polygons, whereas MacPaint handles pixels only. Suppose a user performs the following actions: draw a circle C with MacDraw, cut C from MacDraw, paste C into MacPaint, cut C from Macpaint and finally paste C back into MacDraw. As far as the naive user is concerned, C looks like a circle in the MacDraw document, but is not editable anymore as a circle. Cut and pasted operations have lost "semantic" information about transferred data.

Consistency in the behavior of "cut and pasted " information relies on the existence of a universal format, as well as on a general type translator.. A universal format defines a common data representation, i.e., a common formalism, for all of the applications, say, of a workstation. A type translator performs the required transformations between the data representations specific to an application and the universal format. To our knowledge, "type recasting" is a research topic that has not been investigated in its full generality.

- Conciseness and Undo and Redo Features

Undo has two advantages: it allows the user to easily correct a mistake and it avoids the execution of the plan of actions that would undo the

desired effect. A redo feature avoids the repetition of a sequence of actions. Both undo and redo support conciseness.

- Conciseness and Default Values

  Default values are another means to reuse information. There are two kinds of default values: static and dynamic. Static values do not evolve with the session. They are generally wired in the system, or are acquired at initiation time from a profile file. On the other hand, dynamic default values evolve during the session. They are computed by the system from previous user inputs. Figure 2.11 gives an example of the default value proposed by a system for the file name of a document being saved in the course of an editing session.

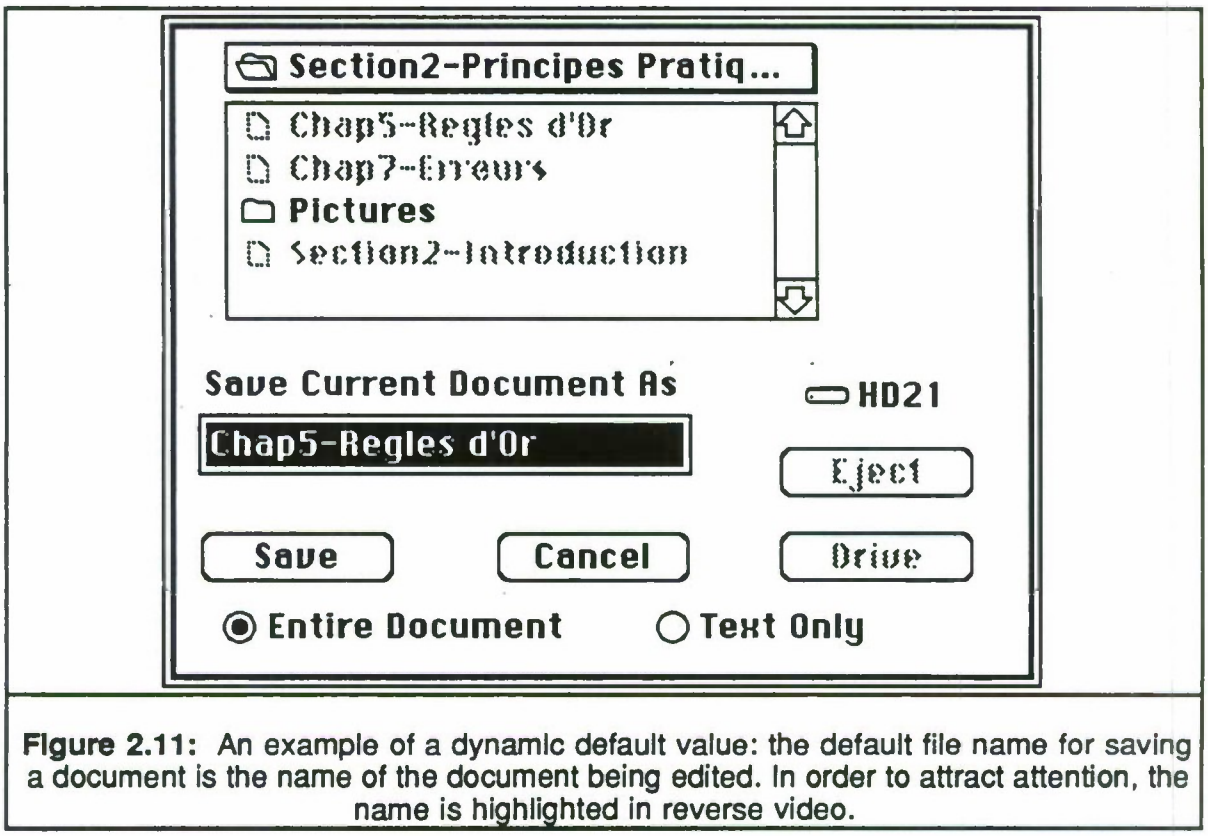


Figure 2.11: An example of a dynamic default value: the default file name for saving a document is the name of the document being edited. In order to attract attention, the name is highlighted in reverse video.

2.6.2.3. Guideline 3: Cognitive Load Reduction

The literature describes many ways of reducing the cognitive load. Among them, we select the use of menus and forms, and the informative and immediate feedback.

- Cognitive Load Reduction and Menus/Forms

Experiments show that the human being is better at recognizing than at recalling. Menus and forms, which present alternatives, are good alternatives as short-term memory extensions.

- Cognitive Load Reduction and the immediate and informative Feedback

  Generally speaking, a feedback is a reaction to some cause. In the context of human-computer interaction, the feedback is an output expression produced by the system that has processed some user input. The interpretation of the feedback by the user leads to the evaluation of the situation before carrying on the plan of actions. Thus, the feedback has the responsibility of expressing the state of the system.

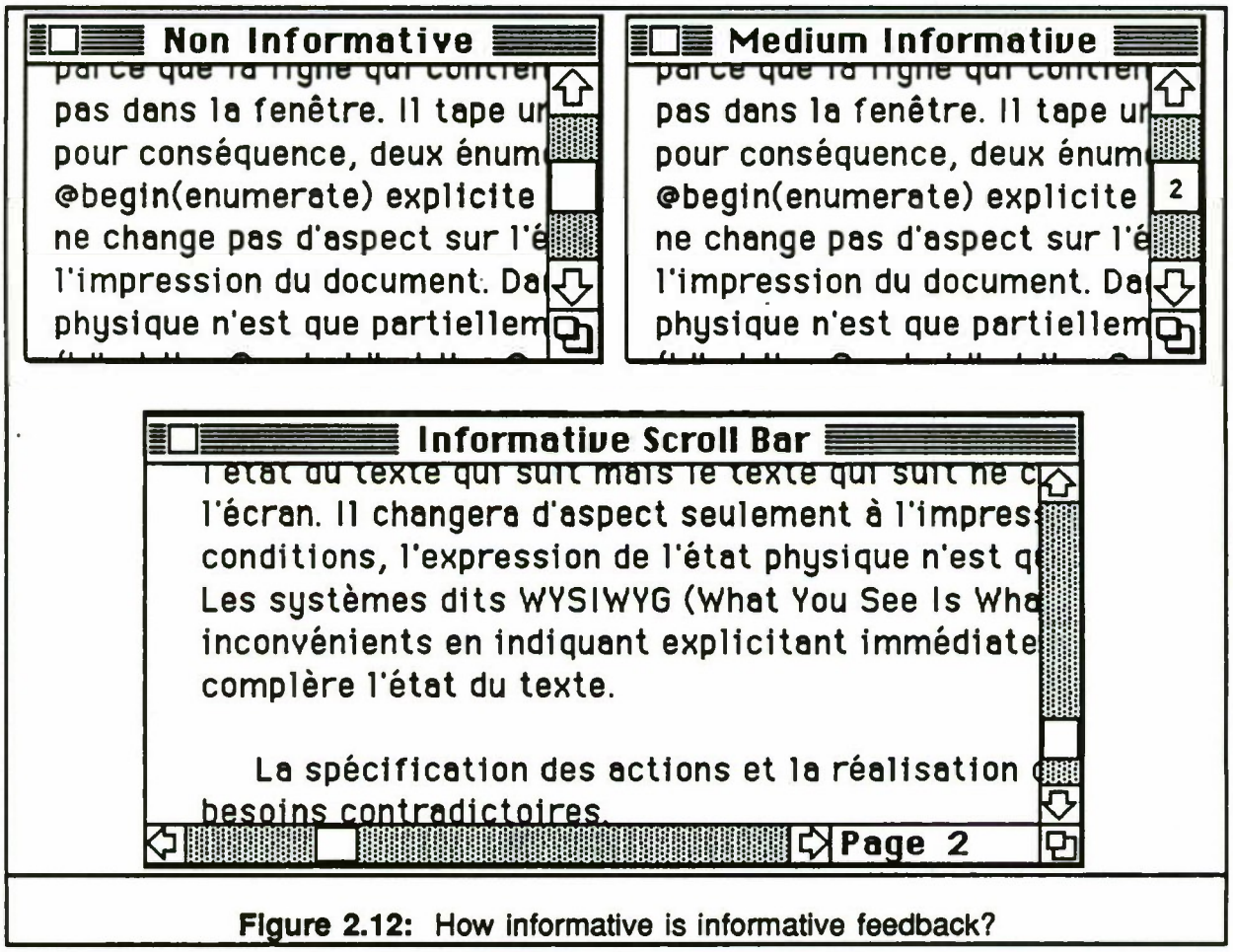


Figure 2.12: How informative is informative feedback?

The system state is described by a wide variety of data structures. As far as human-computer interaction is concerned, the system state is comprised of the data structures that are of interest to the user. These data structures are those that match the psychological variables involved in accomplishing the task. The system feedback has the responsibility of presenting these data structures in a form that helps the evaluation. It is also in charge of immediately informing the user of the changes happening to such structures.

The changing shape of the cursor.is an example of immediate feedback. A cursor shape can be used to remind the user that a particular mode is active (e.g., drawing or erasing); cursor shapes, such as the hour glass, the wrist watch (or the cup of tea) are useful to inform the user of a long operation. Dynamic techniques such as progression bars convey more information about the evolution of a time-consuming operation.

As an illustration of how informative an informative feedback can be, we consider the presentation of the psychological variable "page number," which is of interest in document editing tasks. Figure 1.12 presents three possible feedbacks for this variable. On the top left corner of the figure, the position of the elevator in the scroll bar indicates that the current view is about half-way in the document. On the top right corner, the elevator includes extra information: an integer. After some practice, the user infers that it refers to a page number. On the bottom screen, the user is fully informed of the current position of the window in the document.

In a nutshell, informative feedback should answer the following user questions [Nievergelt 80]: "Where am I?, What can I do?, What have I done?"

2.6.2.4. Guideline 4: User-Driven Interaction

Users should have the initiative in a dialogue with a computer. This recommendation stems from the view of the computer as a tool: the computer is a submissive server, whereas the user is the principal actor. Actually, there is a more generous view of the computer: that of a collaborator.

In a collaboration, each partner acts according to each one's competence. In the particular case of human-computer interaction, the computer should behave as the extension of the user's skills. It should let the user act freely and take control arbitrarily. The difficulty for the user interface designer lies in identifying the transition points where control shifts from the user to the computer and back.

In both cases, whether the computer is a tool or a collaborator, users should not be modeled as finite state machines. Automata offer a convenient way for modeling relations between predictable and well-defined states. States involved in human problem solving are rather unknown and their relations are mostly unpredictable. Human problem solving is basically opportunistic, mixing the top-down approach with the bottom-up approach [Hayes-Roth 79]. As a result, it must not be constrained by an inflexible model of interactions.

To summarize, give the user the illusion of driving the system.

2.6.2.5. Guideline 5: Flexibility

Flexibility is mainly concerned with the notions of customization and multiple rendition of a concept.

- Flexibility and Customization

  Customization is the adaptation of the user interface to the user. A user interface can be adaptative or adaptable. An adaptative user interface automatically evolves depending on the user's mental state. An

adaptable user interface is manually modified to fit the user's requirements. In both cases, whether the user interface is adaptative or adaptable, the current facilities for customization are rather limited.

An adaptative user interface relies on the existence of an intelligent observer that tracks the actions of the user, infers the user's mental state and modifies its behavior accordingly. The notion of intelligent observer supports the view of the computer as a collaborator. Unfortunately, the realization of an effective observer relies on a thorough understanding of human cognitive behavior. Given our limited knowledge in this domain, a lot of research needs to be pursued in the area of adaptative user interface. Currently, a more practical approach is the manual adaptation of user interfaces.

An adaptable user interface relies on the existence of a software architecture that makes a distinction between functional mechanisms from presentation policies. Functional mechanisms implement the high level semantics of the interaction, whereas presentation policies deal with the syntactic and lexical issues. A software architecture that satisfies this requirement makes possible the modification of the syntactic and lexical aspects of the system without side effects on the internal functioning. For example, it is easy to repair the "surface" of the interaction, such as changing a command or a parameter name, without any code recompilation. Although it is possible to modify the lexical and syntactic aspects of the presentation, it is not possible to change the structuring of the interaction. This issue is the topic of Guideline 6.

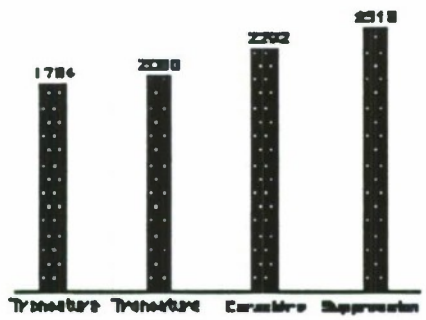
Other complementary approaches to customization include facilities for building new commands (macrocommands) and defining abbreviations. These two aspects have already been discussed in 2.6.2.2.

A priori customization seems to conflict with consistency. In analogy to architectural design, a framework is provided that can be moderately reorganized and decorated as desired: it will be possible to change the location of a secondary wall but certainly not the location of a wall that supports the building. It is also possible to choose wallpaper and carpeting, because it is independent of the framework. Similarly, with an appropriate software architecture, it is possible to change the lexical and syntactic aspects of the interactive system without damaging the overall organization that is the referential framework for consistency.

- Flexibility and Multiple Rendition

  Multiple rendition is a facility for multiple, possibly simultaneous views of a given concept. Each view matches a particular need at some stage of a given task. For example, in text editing, it could be possible to view the document as a table of contents and simultaneously read a particular chapter or subsection. The table of contents and the subsection are two views of the same data structure that represents the document.

Figure 2.13 gives an example of a multiple representation of the same concept. Chapter 4 describes some software techniques that support multiple rendition.



	1164				1252
	2868		365		1391
	2232		625		1457
	2679		1267		1900

Figure 2.13: Multiple representation of the same concept. On the left, data are presented as bar charts; on the right, the same data are collected in a table.

2.6.2.6. Guideline 6: Structured dialogue

Structuring is a general technique for mastering complexity. Dialogue structuring consists of organizing the command space into layers of increasing complexity. By doing so, the novice user is able to successfully accomplish simple tasks that are presented right away in the system image. As the system becomes more familiar, the user will gradually discover new functions more complex to handle but not necessarily mandatory to get usual tasks done. Dialogue structuring into levels of increasing complexity is known as the "training wheels" technique [Carroll 84].

This principle of dialogue structuring, which has the nice effect of leading to successful task closure (feeling of relief, satisfaction of work done), is certainly not easy to put into practice. It requires a thorough task and user analysis which is not often performed by computer scientists.

2.6.2.7. Guideline 7: Error Prediction

Errorless interaction is illusory, but the computer system can provide support for error detection and error recovery. D. Norman, [Norman 86] identifies two classes of errors: mistakes and slips. A mistake results from the formulation of an inappropriate intention. A slip is an unintended action. Both of them, mistakes and slips, generally come from the inadequacy of the system image. The system image should minimize error occurrences, and facilitate error detection and error repair.

- Support for minimizing errors and for improving detection

  Occurrences of errors can be minimized and error detection can be improved in several ways: an appropriate metaphor of interaction, an adequate terminology, and an immediate and informative feedback. When considering slips only, techniques dealing with concision avoid slips by allowing the user to reuse information without any risks of enter incorrect data.

  A metaphor of interaction defines a model to which a novice user can refer by analogy to interact with the system. There are

currently two major metaphors for interaction: the world metaphor and the conversation metaphor [Hutchins 86]. The world metaphor electronically mimics objects of the real world. A popular example of the world metaphor is the desktop metaphor, where icons represent actual folders and documents, and where the mouse is the electronic extension of the hand. The conversation metaphor is based on a linguistic description of the actions to be performed on system objects. Examples of the conversation metaphor include the textual command languages such as the Unix Shell. In the conversation metaphor, the user talks about an implicit world (the user describes what is to be done), whereas, in the world metaphor, the user directly manipulates objects (the user does not tell how to do it, but does it instead). Thus, "direct engagement" of the user shortens the gulf between mental and computerized representations. It should minimize error occurrences. However, in cases where there is a mismatch between the metaphor and its electronic implementation, errors might be created rather than reduced. Consequently, care should be taken to make clear the limits of the metaphor used.

Adequate terminology has to do with the choice of names. Consistency is an important feature in naming but the terms should be understandable to the user. Well designed software architecture combined with tools for lexical and syntactic customization can overcome an inappropriate wording.

Immediate and informative feedback has been discussed in 2.6.2.3 in relation to reducing cognitive load. With regard to errors, feedback may avoid slips such as forgetting the current mode of interaction. It may protect the user from making wrong decisions or wrong inferences.

- Support for Easy Repair

Error repair is a problem solving activity. The support for such activity comes in several forms. It includes undo/redo facilities and informative error messages.

The combination of undo and redo facilities, not only avoid possible slips during the respecification of a command, but also encourage investigation. As such, they provide the user with an effective support for problem solving.

Error messages, such as "SYNTAX ERROR!," are useful for error detection but are far from being helpful for error repair. They require a rather fastidious and sometimes frustrating evaluation phase. Error messages should clearly express the exact cause of the error and provide the user with additional information about state variables relevant to the current problem. Figure 2.14 illustrates the case of an error message helpful for error repair.

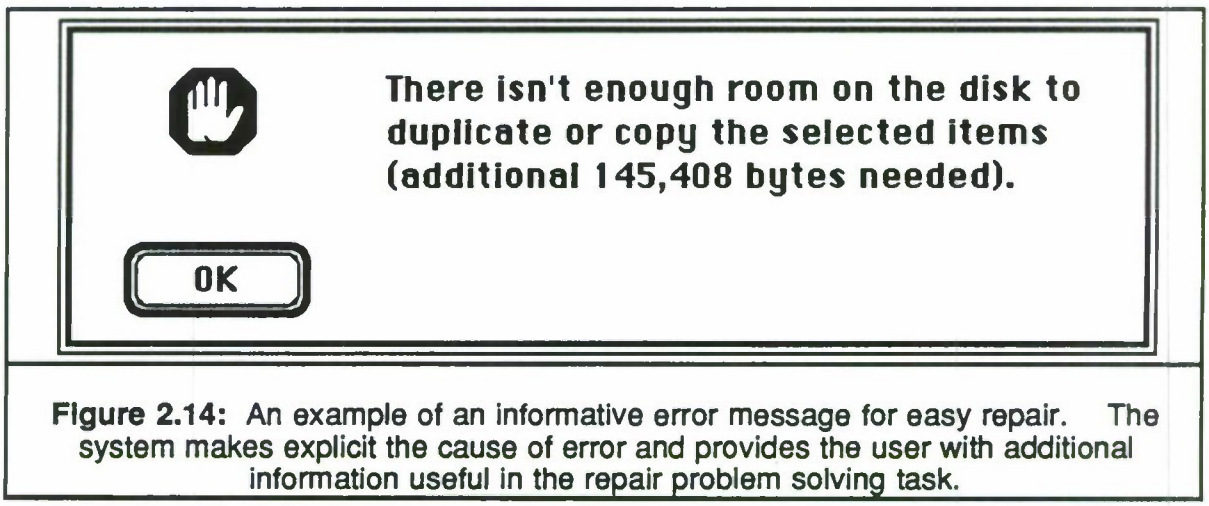


Figure 2.14: An example of an informative error message for easy repair. The system makes explicit the cause of error and provides the user with additional information useful in the repair problem solving task.

3. The Levels of Abstractions in Interactive Software

This chapter:

- identifies the abstractions involved in constructing interactive software.
- introduces the terminology that will be used in the remainder of this document.

3.1. Introduction

An interactive system calls for various levels of services, ranging from low-level physical I/O handling to the high-level management of the interaction. As shown in Figure 3.1, these services may be viewed as a hierarchy of abstract machines:

- At the bottom of the hierarchy, device drivers directly control the physical devices. A device driver is a program tailored to the physical functioning of a particular class of devices. Interactive software includes a driver for each class of devices it supports. Generally, these drivers are part of the underlying operating system. They define the device dependent layer.
- The next layer hides the diversity and the functioning of the physical devices by defining a virtual terminal. A virtual terminal provides client programs with device independence but is not able to support device sharing.
- Device sharing between multiple software activities is implemented by window systems. Window systems give client programs the illusion of being the unique owners of one (or several) virtual terminal(s). Virtual terminals are programmable at a fairly low level of abstraction. This level may not be convenient for client programs which deal with highly structured data.
- Abstract image machines shorten the gap between the internal representations used by client programs and the external representations required by the graphics package provided by (or sitting on top of) the window system. An abstract image is an intermediate data structure which expresses output rendition at a high level of abstraction and which supports high level input facilities. Inputs and outputs, whether they are expressed at a high level of abstraction or not, require some kind of control that organizes their occurrence.
- dialogue control shapes the interaction between the application and the user down the way through the underlying abstract machines. The dialogue machine can be seen as a mediator between the application and the user. It bridges the gap between the abstract, media independent world of the application and the universe that makes up the user interface.
- At the very top of the hierarchy, the application implements the functional core of the interactive system. This core is media independent, that is,

- At the very top of the hierarchy, the application implements the functional core of the interactive system. This core is media independent, that is, has no knowledge of the way its data structures and functions are exposed to the user. Its purpose is to implement an expertise in a specific domain that will allow the user to perform specific tasks in that domain. It is not concerned by how this expertise is made accessible to the user.

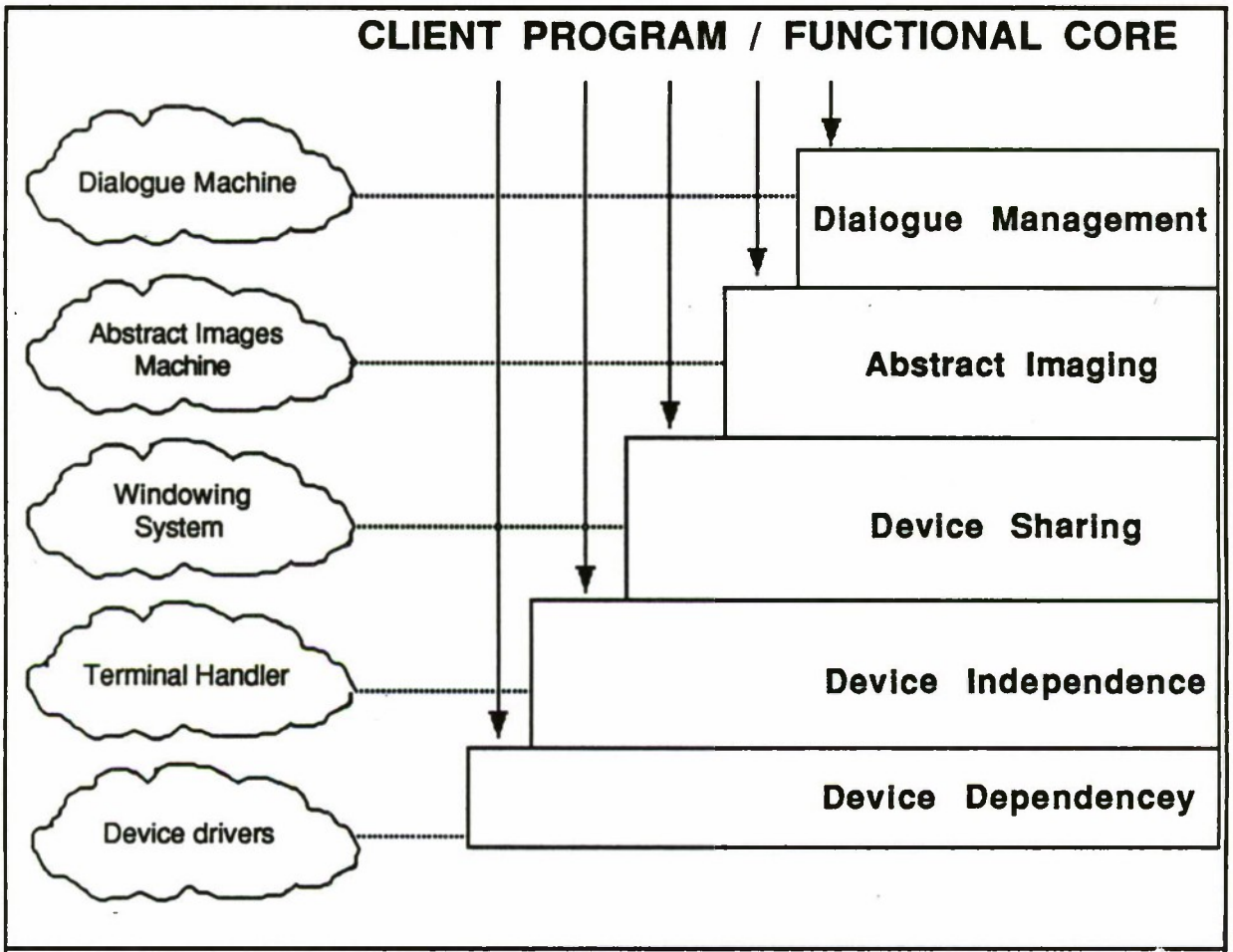


Figure 3.1: The levels of abstraction involved in the construction of interactive software.

The following sections detail the nature of the abstractions that respectively allow for device independence, device sharing, abstract imaging and dialogue management.

3.2. Device Independence

Physical independence has multiple facets. It primarily comes in the form of a virtual terminal that hides the actual functioning of quite different I/O devices without modifying client programs. It may also allow for the addition or suppression of new devices without recompiling or even relinking the existing code. In this section, the focus is on code reusability. We first identify the problems due to the diversity of physical devices. We then sketch the principles of how physical independence is achieved.

3.2.1. The Problem

Reading or writing data with direct control of the physical terminal presents two difficulties: first, it imposes a precise knowledge of the functioning of the physical devices on the programmer; second, and more important, it compromises software portability. For example, to move the cursor to "line1,column2" of the physical screen, a programmer would provide a VT100 driver with the sequence "ESC[1;2f." Clearly, this sequence becomes obsolete when the VT100 is replaced by a bitmap display.

The software solution to the diversity and the complexity is the use of the abstraction mechanism. In the case of interest, the abstraction is a virtual terminal which provides client programs with a unified and a simplified view of actual terminals.

3.2.2. The Notion of Virtual Terminal

A virtual terminal is an abstract terminal. As such, it provides client programs with an instruction set for expressing inputs and outputs, and the instruction set can be mapped to a variety of physical terminals. Let's see the principles of these I/O primitives.

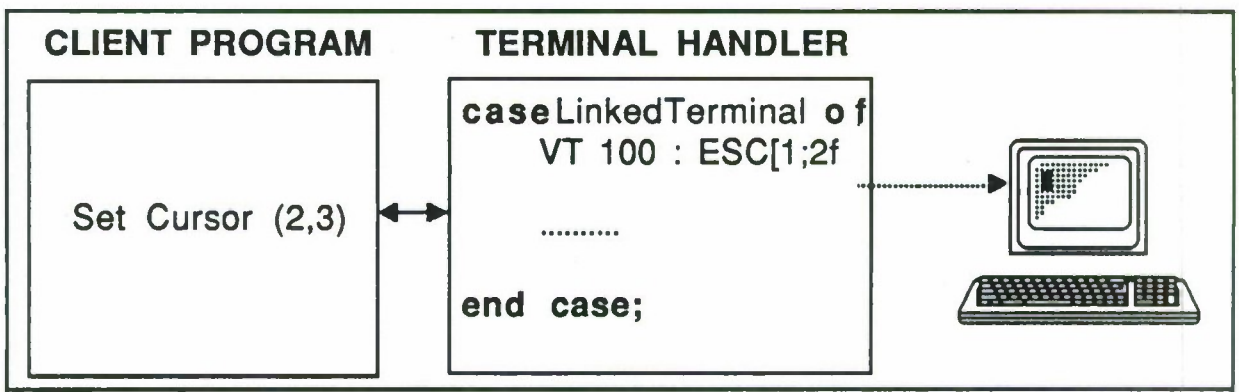


Figure 3.2: The principles of output operations of a virtual terminal.

Figure 3.2 illustrates the principles of output operations of a virtual terminal. In this example, the primitive SetCursor issued by the client program moves the cursor to location (2;3) in the virtual space coordinate. The virtual terminal, whose job is the interpretation of primitives from client programs, translates the virtual location into the

physical coordinate space and calls device dependent primitives. These primitives correspond to the physical device that is currently linked to the client program.

This example mentions the notion of space coordinate. Virtual space coordinates may be integer or fractionary systems. The choice between the two is based on compromises between ease of implementation and effectiveness of physical independence. As an example of compromise, the virtual terminal defined in X-Windows is based on the hypothesis that physical screens have square pixels. As a result, the primitive that is supposed to draw a circle produces an ellipsis on screens whose pixels are rectangular (such as the Apple Lisa or TV screens).

For inputs, the chaos due to the diversity of physical devices (keyboards, mouse, electronic glove, etc.) has been organized in the form of typed classes. The types are specific to a particular implementation of a virtual terminal. In general, they include:

- The class *key*, which models physical keyboards.
- The class *locator* to denote the location of a pixel in the virtual coordinate space.
- The class *choice*, which returns an integer value useful to represent mouse buttons.
- The class *valuator* to model physical devices such as potentiometers that generate real values.
- The class *modifier*, a bit string whose value can be interpreted as a modifier of the semantics of the value returned by other classes.
- The class *application* to allow client programs to synthesize client-dependent events.
- The class *time-stamp* to indicate the time when a physical action happened.

Input classes such as locator, choice and valuator, were first introduced by GKS [ISO 85] and Core. Today, they are implicitly embedded in the device-independent layer provided by window systems. Other classes, such as modifier, application and time-stamp classes have been made popular by window systems. The last two deserve additional comments:

- The *application* class allows client programs to extend the basic set of input classes. Application programs can set up their own protocol of communication by defining special purpose events, and exploit the communication mechanism provided by the window system. This feature is an interesting property of the Macintosh event manager.
- The *time-stamp* class is useful to overcome two types of hardware limitations. The first limitation is the sequentiality of the interrupt mechanism: two actions that appear as simultaneous to the user are reflected to the client program as two separate events. A time-stamp value may be considered a means to glue the events back into a single

abstract one. The second limitation happens at the physical device itself. One well-known example is the one-button mouse of the Macintosh, which can be used as a two-button (or even a three-button) mouse by double (or triple) clicking the unique physical button. Again, time-stamps associated with the successive events allow for synthesizing events.

To complete our picture of the functioning of a virtual terminal undertaken in Figure 3.2, we need to observe Figure 3.3. The client program acquires an event through the GetEvent primitive provided by the virtual terminal. This event is a device-independent description of some action performed by the user on physical input devices. The job of the client program is to interpret the content of the description. The job of the virtual terminal is to build the abstract representation of the physical events. In the example of Figure 3.3, the returned event is a combination of a locator and a choice that represents the screen location pointed to by the user with a mouse.

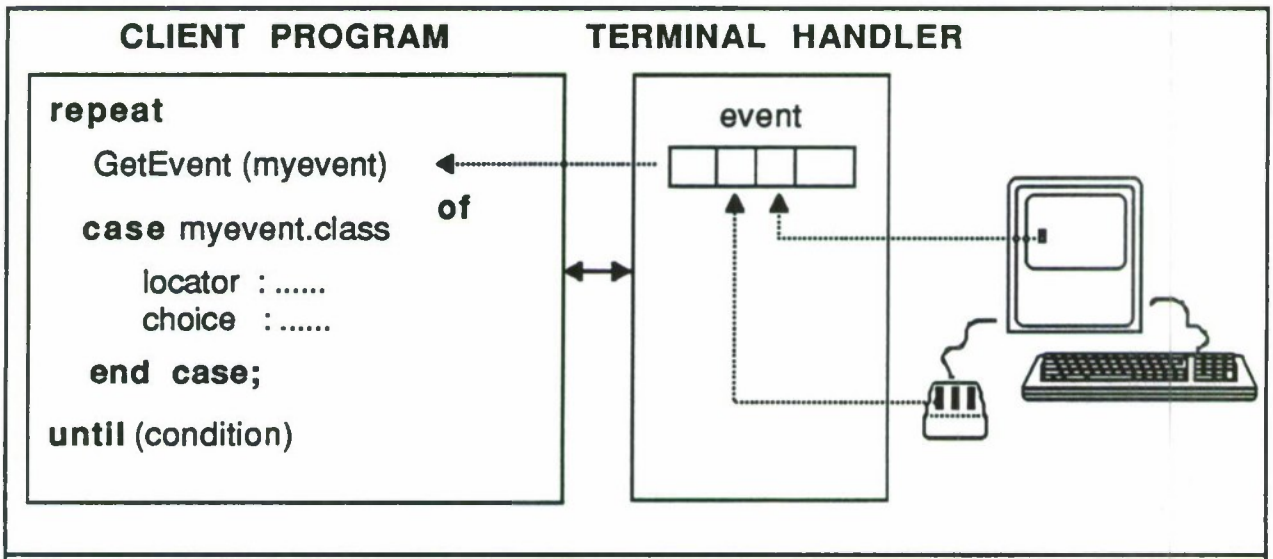


Figure 3.3: The principles of input operations of a virtual terminal

To summarize, device independence is:

- Primarily intended to increase software portability by allowing the substitution of physical devices without damaging existing code. Although this capability is a desirable feature for programmers, it should be stressed that, from the point of view of the user, physical devices are not equivalent. Card, Moran and Newell suggested [Card 83] that the mouse is adequate for the selection of 2D objects, whereas the joystick is more appropriate for 3D manipulations.

- Embedded in window systems.

- Hard to achieve fully.

3.3. Device Sharing

Device sharing is built on device independence. Its purpose is to make available not a single virtual terminal as device independence does, but instances of virtual terminals. Why is this interesting? What are the principles of its realization in windowing systems, and what is the trend in current windowing systems?

3.3.1. Justification

A virtual terminal, as provided by the device independence layer, is a resource that may be simultaneously accessed by multiple activities. In the late sixties, multiprocessing was not accessible to the user. Processes were internal creatures that helped the system do its job. Today, the user can explicitly or implicitly launch multiple activities, all of which act as producers and consumers of the terminal. In the same way that system engineers introduced the notion of virtual resources (e.g. virtual memory) to extend the capabilities of the core hardware components, interactive software engineers defined windowing systems to extend the capability of terminals. Windowing systems behave like virtual resource generators by providing client programs with any number of virtual terminals.

Device sharing is necessary for multiprocessing environments. Whether the environment is multiprocess or not, it is also useful as a technique for organizing information on the screen: output expressions that are linked by some logical criteria need to be physically gathered on the screen. Regions that result from such grouping compete for rendition. This competition also occurs for input events which are to be dispatched to the appropriate destinatary (process, region, etc.). This is the familiar multiplexing/demultiplexing problem that is commonly encountered in operating systems. In the case of interest, the solution to the problem is based on the notion of window.

3.3.2. The Window at the Center of the Multiplexing/Demultiplexing Mechanism

The notion of window and the terminology vary widely from one window system to another. It is necessary to distinguish between the window as the elementary drawing surface that is mapped onto the screen, the notion of drawing surface that needs not be mapped onto the screen, and the window as the object that the user manipulates. The way these notions are implemented and organized together depends on the window system. The primary interest in this section is the window as the elementary drawing surface mapped onto the screen. More details are provided in Chapter 4.

A window as an elementary drawing surface is a drawing context. This context includes a set of pixels and a system coordinate space. The set of pixels is used for rendering output expressions and for returning pixel locations expressed in the window coordinate space. To take advantage of possible hardware support for raster operations, the set of pixels usually forms a rectangular area. This area is conceptually visible on the screen and defines the sharing unit.

Sharing uses the notion of owning: a window has an owner (e.g. a particular process). The way the owner is identified is out the scope of this subsection. Output requests issued by client processes are demultiplexed by the window system; input requests are

multiplexed. For output, the window system clips any information that lies outside the drawing area of a window. Input events are dispatched according to two possible techniques. Either the window system, such as NeWS [SUN 87], broadcasts the event to all of the windows that have expressed their interest in this type of event, or, such as X Windows [Scheifler 86], sends the event to the "current focus window." The fact that a particular window is the current focus for keyboard events or for any combination of typed events is decided by some client process by issuing the appropriate request.

3.3.3. Trends in Windowing Systems

At first sight, window managers look very similar: client programs can create windows, move windows, resize windows, etc. Nevertheless, there is no common terminology; the basic functional concepts such as the way events are dispatched differ profoundly for one windowing system to another; as showed in Chapter 4, windowing systems also differ in their architecture. However, the new generation of window systems illustrated by NeWS and X-Windows have several features in common:

- Presentation policies are distinct from functional services. By doing so, the "look and feel" of windows can be customized without changing the code that implements terminal resource sharing;
- A server is in charge of the execution of the functional services. By doing so, client programs and the windowing system need not be running on the same physical machine and client programs can create remote windows.

These issues will be further discussed in Chapter 4. To summarize the topic about device sharing, we take the point of view of the user. Window systems allow the user to:

- Carry several activities concurrently.
- Gather information that are semantically connected by some psychological or task criteria.
- Ask for multiple views of the same concepts in distinct regions of the screen.

So far, we have identified and described abstractions that make possible the expression of inputs and outputs in a device independent way and without any risk that a client process will damage other processes space. We need now to analyze the information that is carried by these expressions.

3.4. Abstract Imaging

Abstract imaging is a technique for acquiring inputs and for rendering outputs at a level of abstraction compatible with client programs and windowing systems requirements. The problem posed by input and output operations is identified in the next paragraph. The principles of one possible solution is then presented.

3.4.1. The Problem

Primitives provided by window systems for the expression of I/O's are device independent but the concepts they manipulate generally lie at a fairly low level of abstractions: pixels, lines, circles, rectangles, splines are the usual notions. At most, one finds the encapsulation of graphical requests in very much the same way a macrocommand denotes a set of commands: PostScript proposes the notion of path [SUN 87], QuickDraw implements the notions of region and picture [Rose 86], and GKS the notion of segment [ISO 85]. These encapsulations help structuring the output expressions but the operators they allow are very limited in scope. In particular, the entity described in a graphic macro can be rotated, enlarged, moved as a whole but its content cannot be dynamically modified. This restriction is in conflict with the editing nature of interaction.

In Chapter 5, we will describe tools that are more appropriate for this sort of requirement. For now, the principles of the approach is presented in the next subsection.

3.4.2. The Principles of the Notion of Abstract Image

The purpose of an abstract image is to hide the functioning of the virtual terminal. An abstract image is a data structure that acts as a mediator between some client data structure to be exposed to the user, and a real image expressed in terms of some graphics package. The exact nature of abstract images will be made more explicit in the Chapter 5. For now, we limit the description to the principles. Figure 3.4 illustrates the role of an abstract image.

The client program builds an abstract image that represents a domain-dependent data structure. The abstract image is automatically processed by an abstract image machine. This machine generates graphic requests that are interpreted by the underlying graphics package. The "concrete" or real image can be produced either in an offscreen bitmap, or in a window. If there is no windowing system, then the image must be generated on the physical screen. The choice between the two first techniques depends on the facilities provided by the window system.

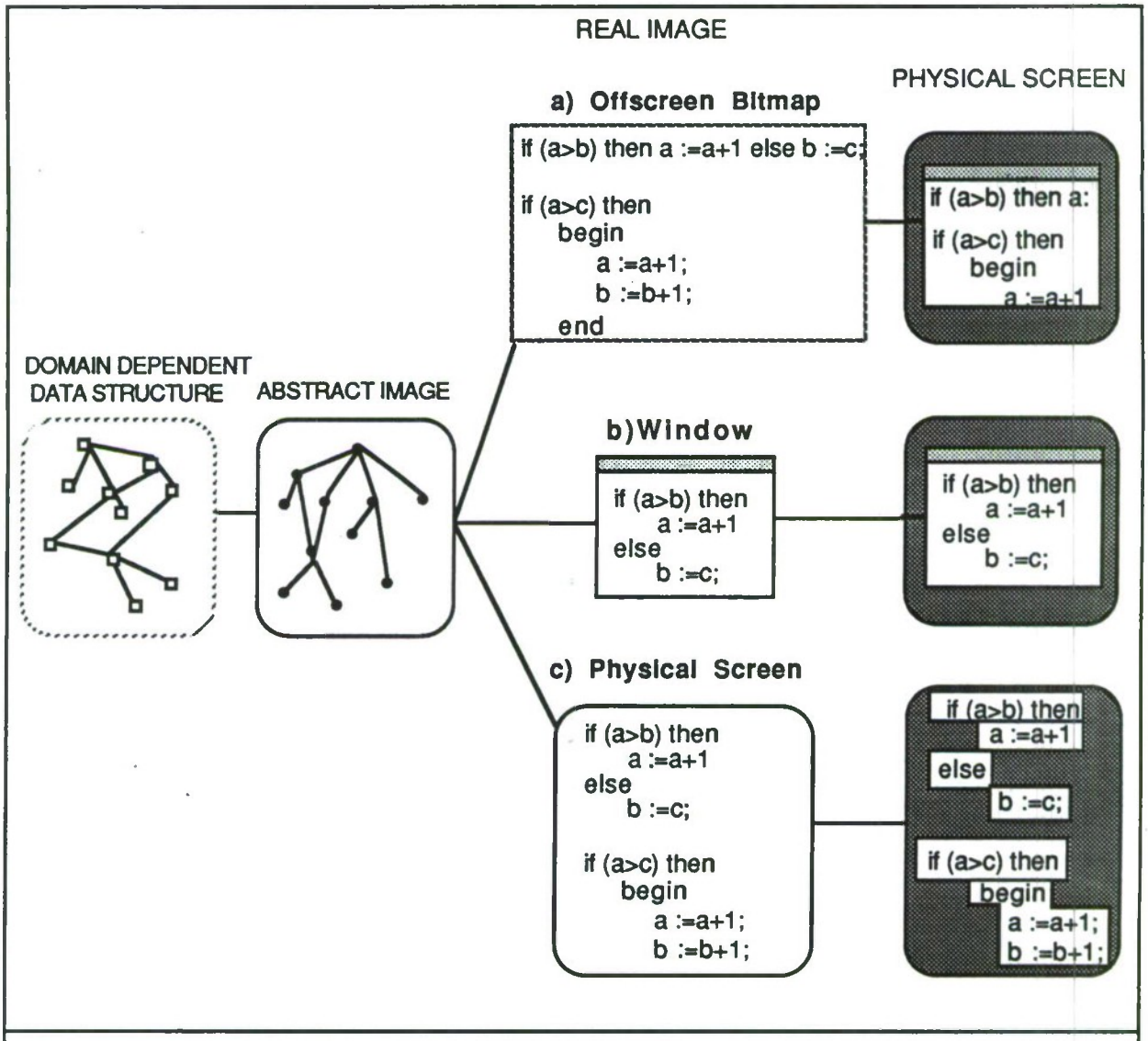


Figure 3.4: An abstract image is a mediator between a domain dependent data structure and a real image.

For some windowing systems such as X Windows-V10, offscreen bitmaps accept a very limited set of operations. In particular, it is impossible to draw on an offscreen bitmap, but it is possible to solely fill it with pixels with raster operations. In such circumstances, the real image must be produced in a window. The advantage of an offscreen bitmap over the direct mapping in a window, is its use as a "visual cache." An offscreen bitmap can be larger that a window. As a result, it may contain extra information useful for repainting the content of an enlarged or scrolled window.

Abstract imaging is not only useful for hiding the functioning of the virtual terminal and for processing syntactic user tasks such as scrolling and resizing windows, but also as a convenient technique for multiple rendition of a given concept. The capability for the user to observe different views of the same concept enhances the flexibility of the interaction. (Flexibility is one of the ergonomics rules described in 1.6.2). Figure 3.5

shows how multiple views are obtained from the same abstract image. In the example, the client data structure represents the concept of a house. its corresponding abstract image is interpreted in two ways. One possible interpretation provides a picture of the house as a floor plan with a lot of details (room names and furniture). in the second interpretation, irrelevant details are suppressed. Both real images may be simultaneously visible on the screen and both are automatically updated as the abstract image is modified.

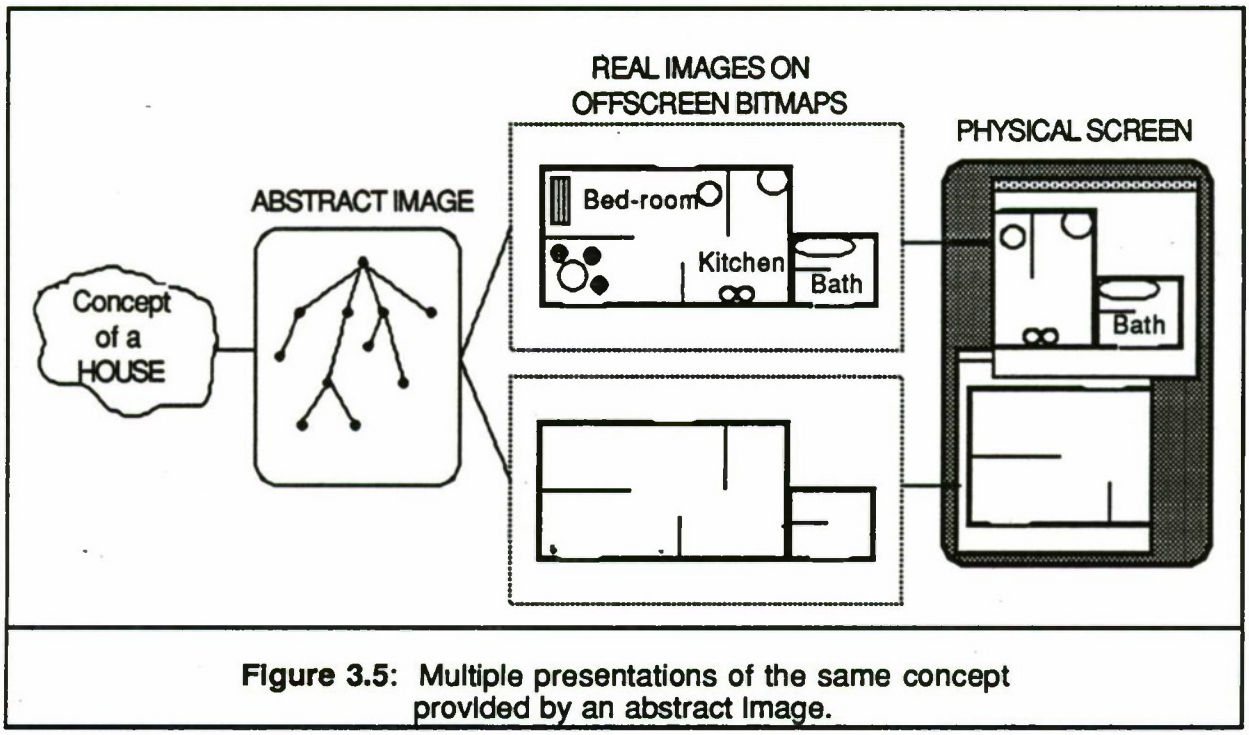


Figure 3.5: Multiple presentations of the same concept provided by an abstract image.

So far, we have described the contribution of abstract images as an output mechanism. Figure 3.6 shows how input is processed: from the selection of a point on the screen to the concept of the client program. Let (x,y) be the location of the point in the screen coordinate system. The selection is first interpreted by the windowing system as a location (x', y') relative to the top window which owns (x,y). The abstract image machine receives a triple which identifies the window and a point (x', y') in this window. The window identification allows the abstract image machine to identify the abstract image, and (x', y') allows it to determine which item of the abstract image owns the selected point. if an item is linked to a concept or a part of a concept, then the client program is directly informed of which concept element the user has selected.

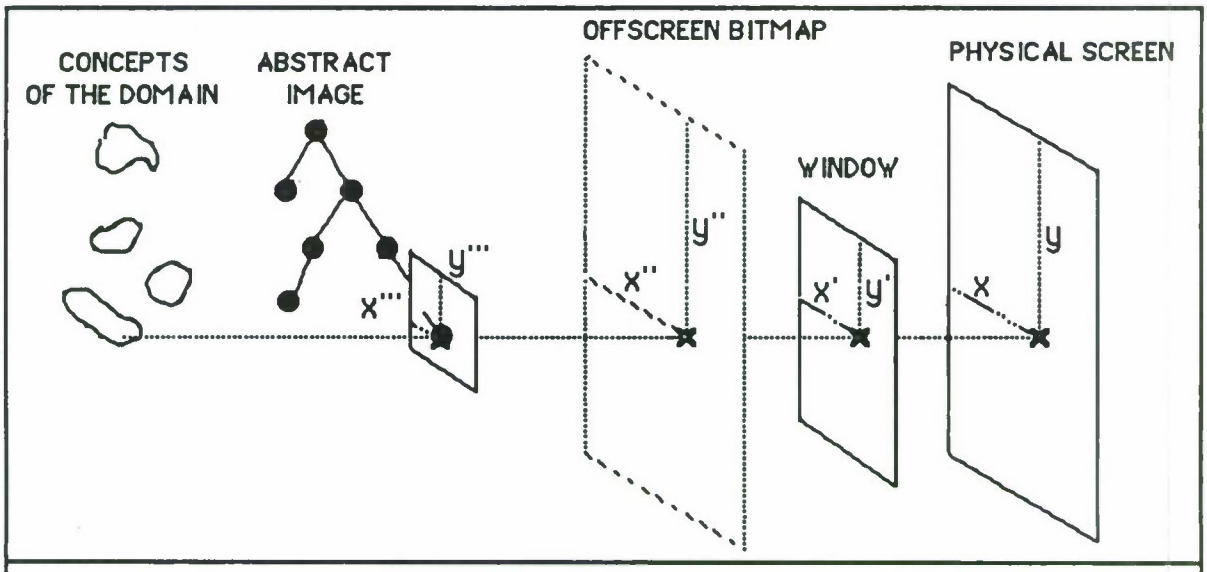


Figure 3.6: From locators to concepts with an abstract image.

While abstract imaging automatically translates low-level input information into client dependent concepts and takes care of multiple rendition and window resizing and scrolling, abstract imaging does not control the interaction between the user and the application. This task is the purpose of dialogue handling introduced in the next section.

3.5. Dialogue Handling

Dialogue handling is concerned with the control and the maintenance of the interaction. This section introduces this issue by making the analogy with actual dialogues between human beings. Dialogue handling is then discussed from the computer side, stressing the fact that the responsibility of the interaction must be shared between the application and the user interface itself.

3.5.1. Introduction

In a conversation between individuals, the control of the dialogue is distributed among the partners. At some point in time, one of the partners initiates the dialogue by submitting an expression or a sequence of expressions to interlocutors. The expressions are processed by the partners and new expressions are produced as results of the processing. Expressions are not elaborated by chance. Their meaning and their syntax depend on the mental representation that each partner maintains of interlocutors in the dialogue.

The interaction between a computer system and a human being should be organized in a similar way. The control of the dialogue should be ruled according to the respective competence of the user and the computer (see Rule 4 about user driven interaction in Section 2.6.2.4). The user makes use of a conceptual model that gathers semantic and syntactic knowledge about the functioning of the computer system (see the definition of conceptual models in Section 2.3). In short term memory (see Section 2.1.4), the user maintains the state of the interaction. Similarly, the computer system

maintains a conceptual model as well as the state of the interaction. As mentioned in Section 2.6.1, CLG offers the designer a convenient way for representing the conceptual model and the state of the interaction with the conceptual and the communication components.

The conceptual component describes the concepts and operations that can be handled by the user, whereas the communication component deals with their presentation. When considering the practical business of designing a software architecture, the conceptual component is naturally mapped into a software component called the application whereas the communication component constitutes the user interface itself. Figure 3.7 shows a simplified view of the software architecture of an interactive system. Given this view as a basis, dialogue handling is handled partially in the application and partially in the user interface.

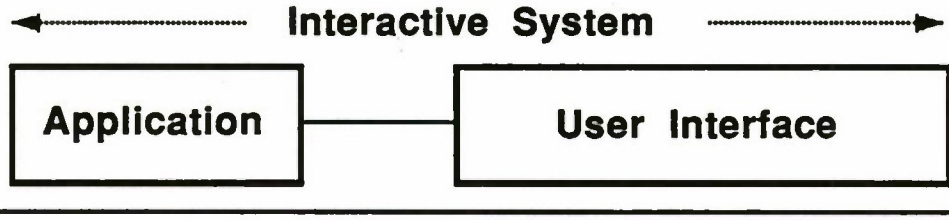


Figure 3.7: A simplified view of the software architecture of an interactive system.

3.5.2. Dialogue Handling in the Application

In the application, the conceptual model is comprised of a set of domain-dependent abstractions that allow the user to accomplish domain specific tasks. These abstractions are data structures and operations. This is a static view of an application. The dynamic view is concerned with the way states of the application relate to each other. A state is the model that an application has for the interaction. It includes:

- The conditions which describe its relations with other states.
- The set of abstractions that are accessible to the external world.

For the application, the external world is the user interface: the user interface is its only partner. The application receives requests from the user interface when its data structures need to be accessed; it sends output requests to the user interface to express changes about its state and data structures. As device independence is guaranteed by windowing systems, so low-level details of the user's actions are hidden from the application.

3.5.3. Dialogue Handling in the User Interface

In the user interface, the conceptual model and the state of the interaction are maintained in a set of agents specialized in human-machine interaction. These agents are mediators between the abstractions handled by the application and the actions of the user. Each one takes part to the interaction. Each one is a miniature interactive

system which handles a piece of the conceptual model and a piece of the state of the interaction. A judicious collection of such active agents defines an instantiation of a user interface for an application. Considered as a whole, a user interface is a translator between the formalism recognized by the application and the formalism employed by the user. At the opposite of the translation process involved in a virtual terminal, the translation process involved in a user interface is very difficult to achieve.

Translation between formalisms for terminals rely on well understood techniques and theories such as finite state automata. The translation process is easy to formalize because the functioning of the source and the target agents are well defined. In the case of human-computer interaction, our knowledge about human behavior is rather fuzzy. However, we do know that human behavior is not well modeled by deterministic computer science techniques. It is not surprising then that the construction of user interfaces is an active area of investigation. Tools for implementing user interfaces are being progressively made available. Such tools are the topic of Chapters 4, 5 and 6.

4. Windowing Systems

4.1. Introduction

Modern computing systems have multiple simultaneous processes ongoing, each of whose processes might have some interaction with the end user. Each process hides its interaction from other processes. The hiding is accomplished through the use of virtual terminals. Chapter 2 introduced the abstractions associated with the notion of a virtual terminal. Multiple virtual terminals all sharing a single physical terminal require management of the terminal's resources. A window sytem is a resource manager for the resources associated with a particular physical terminal. This section discusses some of the issues associated with that resource management, and then discusses an experimental method of managing the complexity associated with multiple windows.

The resources that a real terminal is assumed to have and which are managed by the window manager are:

- High resolution screen. The screen can be bitmapped, rastor or vector.
- Keyboard.
- Pointing device. A multibutton mouse is the most common pointing device, but joysticks, track balls and various gesturing devices also exist.
- Graphic context. The color map for a particular terminal determines which bit patterns represent which colors. The graphic context determines other stataic information such as style and thickness of lines.

4.2. Virtual Terminal

As introduced in Section 3.2.2, Figure 4.1 gives a picture of a single client interacting with a physical terminal. The client provides, at some level of abstraction, images that are displayed on the screen and handles, again at some level of abstraction, inputs that come from the keyboard and the mouse. As a way of making concrete the hierarchy of abstract machines introduced in Chapter 3, consider the user action of selecting an image on the screen. Since at this point we have a client interacting directly with a physical device, the virtual machines that are of concern are the device driver and the terminal handler.

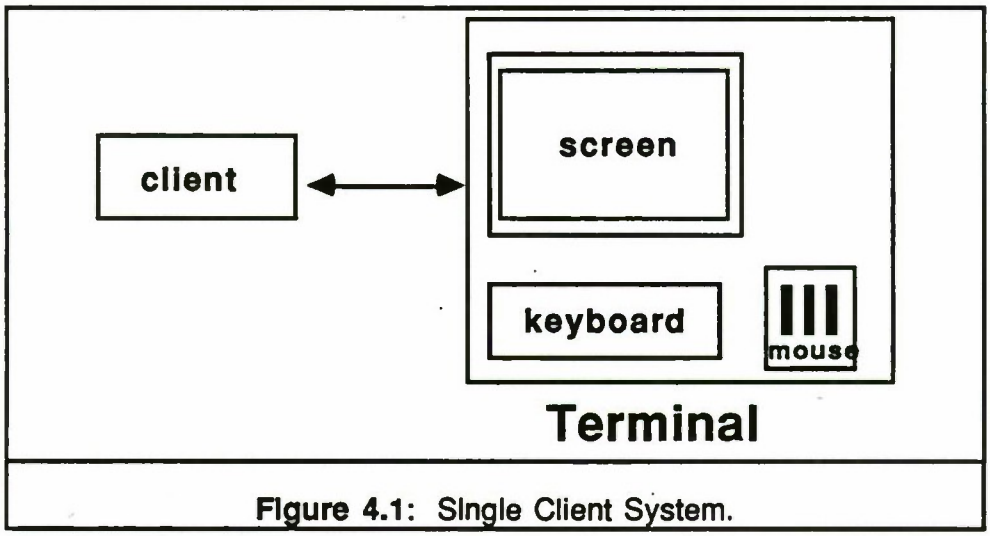


Figure 4.1: Single Client System.

The current cursor position is displayed through some image on the screen. The user moves the mouse. With each increment of movement, the physical controller generates a message to the device driver software. This software calculates the current pixel location of the mouse and reports the location to the terminal handler. The terminal handler generates instructions to move the cursor image to a new position on the screen and passes those instructions to the device driver which generates the new bit map to be displayed. When the user performs a button down, an interrupt is generated, the interrupt is passed through to the terminal handler. The terminal handler then informs the client of a button down operation that occurred at a particular location on the screen. When the button is released, the terminal handler is informed of another interrupt and, in turn, informs the client of a button up at a particular location.

Note here several themes which will reoccur. The first is that the feedback associated with the movement of the mouse and reflected in the movement of the cursor is handled by the terminal handler. The second theme is the level of abstraction reflected in the button events. The location of the cursor is hidden by the terminal handler and is reported to the client only in association with another event. Another example of the level of abstraction of the terminal handler is that it does not deal with objects on the screen or with interpretation of events. The mapping of the cursor position into a particular object and the interpretation of the button down, button up as a *select* are all handled at a higher level of abstraction than the terminal handler.

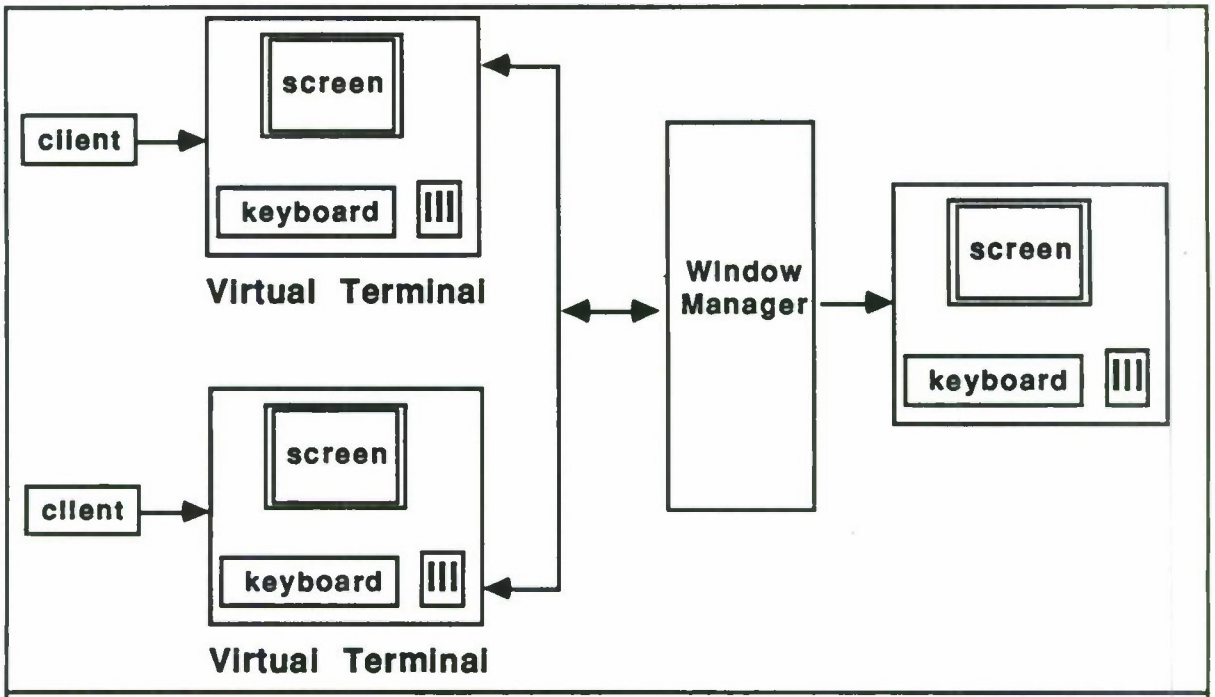


Figure 4.2: Multiple Client System.

Once the client becomes one of a collection of clients, then the real terminal becomes a virtual terminal. The level of abstraction managed by the virual terminal handler is the same as with a real terminal, but the virtual terminal handler must map the multiple virtual terminals onto the single real terminal. The common name for this level of abstract machine is *window manager*. Figure 4.2 gives a representation for the role of the window manager. Figure 4.3 shows a collection of windows.

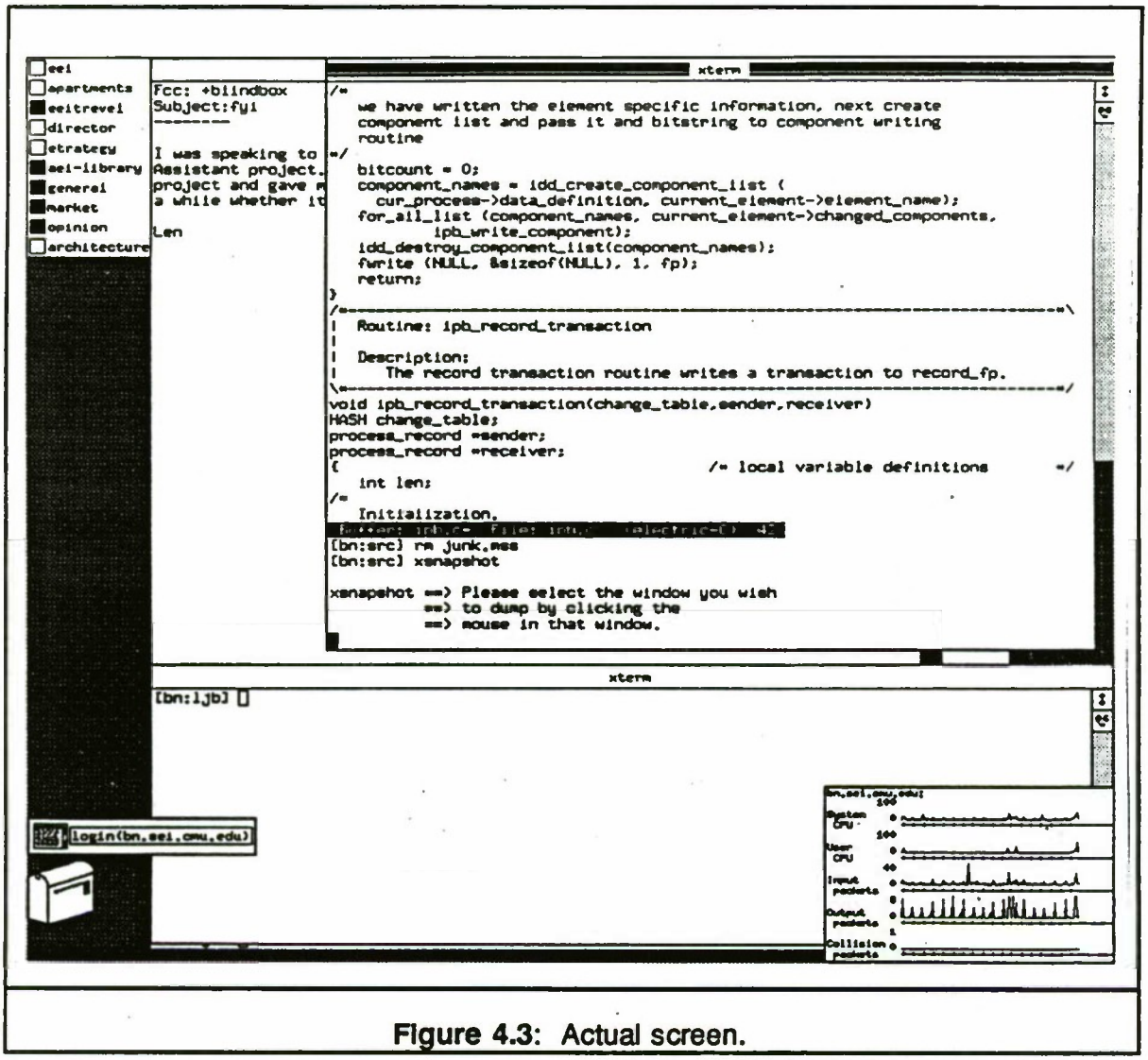


Figure 4.3: Actual screen.

4.3. Single Window

A window is the screen portion of the virtual terminal of a process and provides the output portion of that process. Since the window manager manages the window it is no longer tied to the physical screen size or shape. The window may be represented by an Icon (the lower left corner of Figure 4.3 Is an Icon representing the mailbox used in rural areas of the United States. It represents the output of the mail process). Windows may also have different sizes and locations on the screen.

One of the virtues of abstracting functionality into specific locations is that the functionality can then be embellished without affecting the remainder of the client. In particular within a window system, a window has decorations, geometry, and content.

4.3.1. Decorations

Figure 4.4 displays a single window with its components identified. It has not only the window and its contents, but also it has been decorated with additional functions. These functions are:

1. Title Bar. The window may have a title bar which provides the end user with a cue as to the process that owns the window. The size and title within the title bar can be set by the client.

2. Close Tabs. In the lower right hand corner is a box that enables the end user to iconify the window. That is, when the end user selects this box, the window is turned into an icon by the window manager and additional action is required to expand the window again.

3. Scroll Bars. It is possible that all the information that the client wishes to display cannot be placed simultaneously on the screen. The scroll bars allow the end user to navigate over the whole screen and display the portion desired. This point is further explained in the section on geometry.

Note an additional consequence of performing the abstraction. The original motive behind providing the abstraction was to relieve the client of the management of lower level details. Once the abstraction existed, then it became embellished and the client now has to inform the abstraction manager (the window manager in this case) of additional information to support the embellishments (title for title bar, shape when iconified in the example). The end result of performing the abstraction is that additional functionality is available to the client at lower cost than directly implementing that functionality but the use of the implementation of the abstraction is not free.

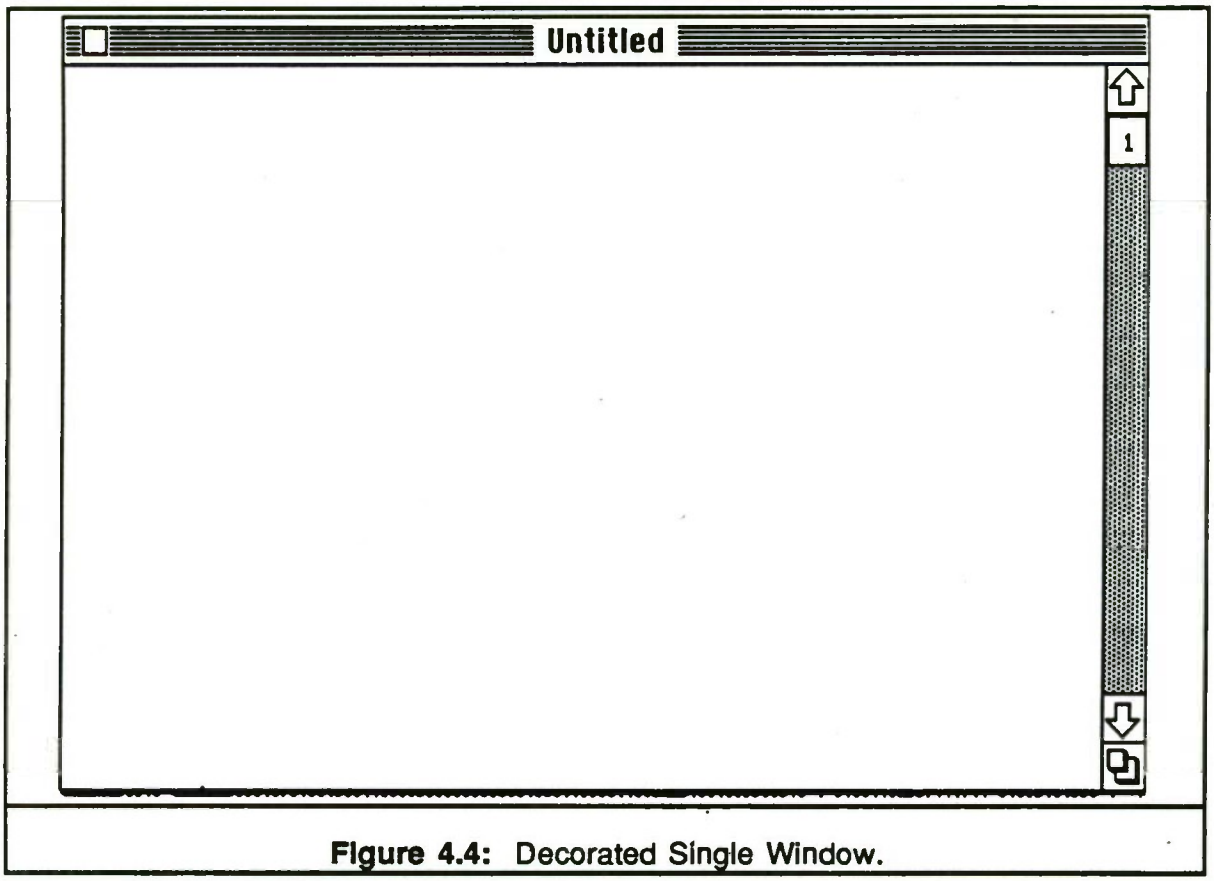


Figure 4.4: Decorated Single Window.

4.3.2. Geometry

The client interacts with a virtual terminal with a single screen size. One of the functions provided by the window system is the *resizing* of the window. The end user may indicate to the window system that a particular window is to be resized and then indicate the new size. The problem then becomes how to map from the size that the client assumes to be the size visible to the user. Three options are available.

1. Display only a portion of the client screen (viewport).
2. Resize the contents to fit the visible window.
3. Report to the client that the visible window has changed size and allow the client to control the display.

4.3.2.1. Viewport

Figure 4.5 displays the situation when a resize has occurred and the resulting window is smaller than the client's virtual terminal. The client has a collection of information, a portion of which has been sent to the virtual terminal. The information available to the virtual terminal represents a *canvas* of information (or an offscreen bitmap). The information available through the window is a viewport onto the canvas. The information is maintained on the canvas using the same scale and proportions as the information sent from the client.

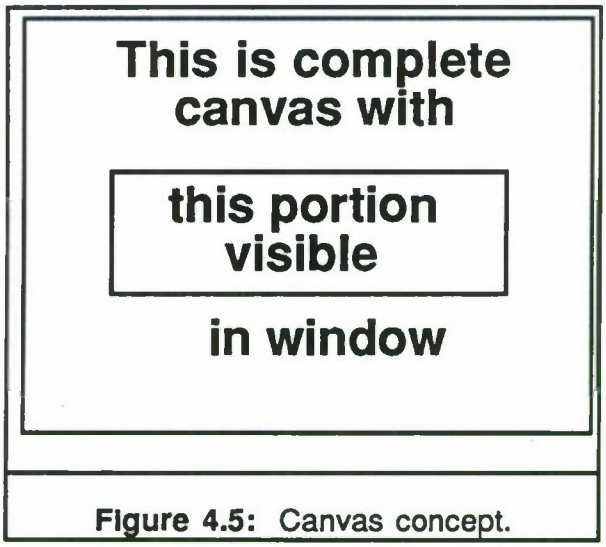


Figure 4.5: Canvas concept.

The viewport can be moved around on the canvas presenting the user with different visible portions of the canvas. This moving around is controlled, typically, with the scroll bars on the viewport window.

The distinction between the information that the client thinks is visible (the canvas) and the information that is actually visible allows the client to generate output to the portion of the canvas that is obscured. The two alternatives when this occurs are to block the client until the information becomes visible to the end user or to allow the output to be placed (logically) on the obscured portion of the canvas.

Note that the size of the information on the canvas does not change when a resize occurs. Only the portion of the information visible to the end user changes.

4.3.2.2. Resizing Contents

Another option when resizing a window is for the window manager to maintain the same information visible to the end user. In this case, the scale of the information must be changed. Pixel replication or sampling techniques are used to expand or shrink the view. Handling aspect ratio changes (the ratio of the sides of the window) becomes a very difficult problem and is typically not dealt with by the window manager. For example, if a circle is displayed in a window and the resize extends the *x* direction without modifying the *y* direction, stretching the image to fill the new window will result in the circle being displayed as an ellipse.

4.3.2.3. Informing Client

Informing the client that a resize event has occurred is the final option for the window system. This option can be used in conjunction with the other two. For example, suppose the viewport becomes larger than the underlying canvas. This client may wish to enlarge the canvas and the window system has no knowledge of how this is to be accomplished.

4.3.3. Shape of Windows

In most systems, windows are rectangular. This simplifies the management of the windows and the clipping of the information within the canvas to the window. On the other hand, rectangular windows make certain selection and display problems difficult. For example, two diagonal lines become difficult to separate with rectangular regions.

In at least one system (NeWS) it is possible to have an arbitrarily shaped window. The boundary of the window is represented by spline curves and the canvas is clipped by the curves.

4.4. Multiple Windows

Window systems manage multiple virtual terminals. This gives the end user a view of the physical screen such as that displayed in Figure 4.3. The management of the resources of the physical terminal involves both the input portion of the terminal and the output. In general, the problem is to allow the end user to differentiate between the various active processes and provide input to the processes as desired.

4.4.1. Input Management

The physical terminal being managed has two different types of input devices. These are the keyboard and a pointing device (a mouse is assumed). Each of the devices generates events of the classes discussed in Chapter 3. The terms key event and choice event will be used to refer to the actions of the key class and the choice class. The basic problem of the window manager is to direct the various events to the appropriate client process. Since each window is assigned to a particular process, this is equivalent to directing the events to the appropriate window. To assist the end user in determining the current state of the window manager two different types of cues are used—the mouse cursor and the text cursor. Each provides the location of one of the types of input devices. Together, they determine to which active process an input event is directed.

4.4.1.1. Mouse Cursor

The mouse is assumed to have a single position within the physical screen. The location of that position is displayed to the end user by means of a mouse cursor. The shape of the mouse cursor can be different depending upon context allowing processes to give the end users a general cue as to the activities of the process.

The position of the mouse cursor is maintained by the window manager level and is available to the client upon request or when a choice event (button press) occurs. The client can also move the mouse cursor to a desired position on a particular window.

Choice events are always directed to the window within which the mouse cursor is located. More properly to the process which owns the window. When windows are overlapping then the event is possibly directed to the window which is invisible below the current window (see Section 4.4.4 for a discussion of window ordering). In certain cases, the overlapping windows are designed to support a single cognitive task (a menu, for example) and in this case, it is the responsibility of the top window to pass the event on to underlying windows.

4.4.1.2. Text Cursor

Some windows are created as text windows. This allows them to receive key events. Within these windows is an additional cursor, the text cursor, which indicates current keystroke position.

4.4.1.3. Current Focus

Keystrokes are assigned to the window that is currently the user's focus of attention. Two models exist to determine the current focus:

1. Mouse focus. Keyboard events are assigned to the window within which mouse cursor is located.

2. Click to focus. The end user must explicitly assign keyboard to a window by selecting that window with a choice event. Keyboard events are assigned to that window unless explicitly changed by the end user or by the client.

4.4.1.4. Cognitive Aspects

Both models for assigning keyboard events present problems. If keyboard events are assigned to the window within which the mouse currently resides, end users can shift their focus of attention and forget to move the mouse to reflect the shift. This results in input being directed to the incorrect process (from the end user's perspective).

The same problem occurs within click to focus systems. The end user can shift the focus of attention without performing the actions required to inform the system of that shift.

One method that systems use to avoid these problems is to give the end user cues which indicate which window is currently the focus. In Figure 4.3, the window in the middle right is the current focus. The text cursor is a square block in that system. The current focus is indicated in two ways. First, the title bar for the window in the current focus is darkened and secondly, the text cursor is filled in within the current window and hollow within the other windows.

Since the window system performs actions for the client (resizing, moving windows, scrolling windows) certain events must be dedicated to specifying these actions. These events then permeate the window system and restrict the types of interactions that a client can specify. For example, if a resize is specified with the right button of the mouse and the client cannot override that specification then the right button is unavailable for the client to use. If the client can override that specification then resize is either unavailable or must be specified in a different fashion depending upon which window is to be resized. This problem is called the *button overload* problem.

One technique used to avoid the overloading problem is to utilize the title bars and scroll bars as areas where window manager functions are specified. If the mouse cursor is within a title bar or a scroll bar then the buttons perform one task and if they are inside a window the buttons perform other tasks.

4.4.2. Output Management

The window manager displays multiple virtual screens on the same physical screen. Typically, all of the active virtual screens will not simultaneously fit on the physical screen. This leads to the problem of the arrangement of the windows on the physical

screen. Figure 4.3 shows seven different active windows. Two of these (the two in the lower left corner) are represented by Icons. Three are text windows (the ones in the middle of the screen) and two are graphic windows (in the upper left and lower right corners). The particular arrangement of windows obscures portions of some of the active windows. The issues involved in output management are:

1. Window placement
2. Management of obscured windows
3. Hierarchy of windows
4. Graphic contexts
5. Data interchange between windows

4.4.2.1. Window Placement—Overlapping

One strategy for the placement of the window on the physical screen is to allow overlapping windows. This is usually associated with allowing the end user to specify the placement of windows. The client generates a window in a particular location and with a particular size and the user then has the ability to move and resize the window. The user also has the ability to make windows visible.

The basis for managing overlapping windows is to maintain a list of active windows. Each window has a size and physical location. The windows are placed on the physical screen in the reverse order of the list. Those windows on the top of the list then become the ones displayed last and, consequently, become the visible windows.

There are two operations available to manage the windows on the list (other than the create, delete operations). These are: move to top of list and move to bottom of list. Move to top of list makes a particular window visible and move to bottom of list removes a particular window from its visibility (assuming there are windows being obscured by the particular window). The window system has a mechanism to allow the end user to specify those two types of events.

The window system also has a mechanism for iconifying and de-iconifying a window. The iconification will not change the position of the window on the screen but will usually cause it to take up less space on the physical screen and make visible other windows.

4.4.2.2. Window Placement—Tiled

A tiled window manager is responsible for the size and placement of the individual windows. The rationale for such systems is:

1. Screen real estate can be more efficiently and more simply managed by the system than by the end user
2. If the end user can only see a portion of a window then that portion should define the client's virtual terminal and since there are no obscured windows, the problems of output to obscured windows do not exist.

Within a tiled window system, each client defines the minimum and maximum window size for a virtual terminal. When less than the minimum is available, the process is suspended. The output from a process is mapped directly into the available virtual terminal.

Tiled window systems will shift the location and size of a window when new windows are created. This can be disconcerting to the end user. Evidence on receptiveness of end users to tiled window systems is mixed. It does seem clear that massive and frequent screen reorganizations, unless user initiated, are undesirable [Bly 86].

4.4.3. Management of Obscured Windows

Output occurs to virtual terminal regardless of window visibility. Windows are also obscured by being overlapped by other windows. This leads to the problem of redrawing the window which is newly exposed. There are two techniques for dealing with exposure of obscured windows:

1. Generate "exposed" event for the client process. This places the client in charge of redrawing the exposed portion of the window. It simplifies the problem of the window manager and saves window manager storage. If the window manager is to have the ability to redraw each virtual terminal then it must maintain a current copy of each window, whether visible or not. This can be expensive in terms of memory.

2. Maintain virtual terminal in separate buffer which is then mapped onto screen. Performance considerations dictate that a separate "frame buffer" is maintained which is used to do the screen mapping. The separate frame buffer limits the number of virtual terminals which can be managed in this fashion.

4.4.4. Hierarchies of Windows

Up to this point, all of the windows were assumed to be bound to distinct processes and to be independent. This allows one window to be repositioned without any effect on the other windows. For some purposes, windows should be considered to be related and either moved together or constrained not to be moved outside a particular region. Some examples are:

1. Figure 4.6 displays a menu. The items of the menus are, in fact, windows all residing within a parent menu. Because the window system will determine within which window the cursor is located, this formulation is more convenient for the client than treating the menu items as the contents of a single window. If the menu items are treated as the contents of a single window then the client must determine which item was chosen when a choose event occurs. Using the parent, child concept, the window system will do the determination. When the parent window is positioned, all of the items of the window should be positioned relative to the parent window.

2. Figure 4.5 displays the canvas, viewport concept. An easy mechanism for managing this relationship is the parent child. The way it is done is slightly counter-intuitive and relies on the fact that the window system clips a window based on its parent. The viewport is the parent window

and the canvas is the child window. Then the portion of the canvas that is visible is determined by the clipping mechanism applied to the viewport window. Scrolling is accomplished by moving the canvas rather than the viewport.

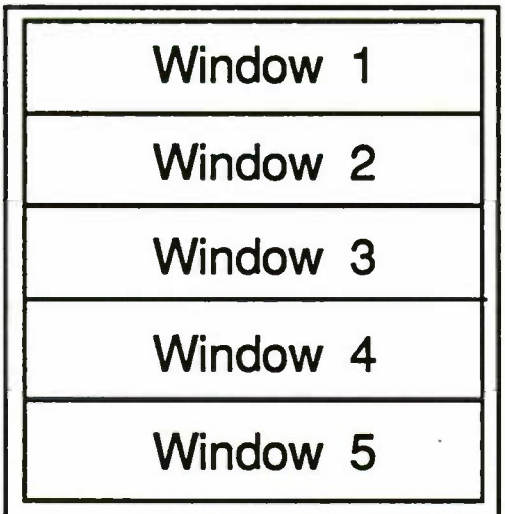


Figure 4.6: Example of menu.

Windows can be specified by the client to form a hierarchy. Within this hierarchy, children are positioned relative to the parent. The children can be moved independently of the parent but the calculation of their position on the screen is done by first determining the position of the parent and then the position of the child within the parent. Children are clipped based on their parents. Thus, when a child window is moved off of the edge of the parent, only a portion of the child remains visible.

A choice event or a key event is directed to the visible window. If that window does not wish to handle the event, it will direct it to its parent, and so on up the hierarchy. All windows are children of the root window and it consumes any unwanted event.

The hierarchy notion allows many complications. Menus have already been discussed. Another use of hierarchies is the title bar, scroll bar concepts that have been discussed. The parent window has children windows which represent the title bar and the scroll bars, etc. Again, this allows the window system to determine the cursor position rather than forcing the client to perform the determination. It, of course, is possible for the client to attach its own title bar, scroll bars to the window and use different mechanism than the window mechanism.

One determining factor in whether children windows are used for auxiliary functions such as menus and title bars is the performance of the window system. Using the window manager for such purposes will generate several hundred windows very quickly. If the window manager is efficient enough to manage a large number of windows then the window abstraction provides for a very attractive solution to choice problems. See Section 5.2.3.3 for a discussion of facilities for manipulating direct manipulation user interfaces.

4.4.5. Graphic Context

The graphic context defines color maps, line style and other graphic attributes (Section 5.3 gives a fuller discussion of graphic concepts). Within a window system, each process has a graphic context and the system typically changes the current graphic context whenever the window focus changes.

4.4.6. Data Interchange Across Windows

Since multiple windows are being managed by the same window manager, it becomes possible to transfer information from one process to another through the window manager. This "cut and paste" facility is implemented by retrieving information from one window (client process) at the level of abstraction of the underlying communication mechanism (see the next section) and communicating that information to a second window (client process). The second process must be able to recognize the structure of the information received but windowing systems automatically have a level of interchanging data from one process to another which is at a higher level of abstraction than pure bit maps.

4.5. Networking Considerations

The functionality of the window manager can be implemented in a variety of different manners. The possible partitioning of the functionality are [Gosling 86]:

1. Replicate the window manager functionality in the address space of each client process.

2. Install the window manager functionality in the kernel of the operating system, outside the address space of the clients.

3. Have a separate window server process which is outside both the kernel and the client address spaces.

The problem with the first option (replication) is the difficulty in multiple processes accessing the same window since the window is maintained in the address space of the process. The problem with the second option (embed in kernel) is that overloads the functionality of the kernel. In order to modify the window manager the kernel must be modified and this introduces configuration problems on most systems. The technique being used by most window systems is the third option. The client processes are considered to be clients of a single server. A number of consequences flow from this partitioning of the functionality.

4.5.1. Communication

Since the client is in a distinct address space from the server, they must communicate through some fixed protocol. The fixed protocol uses the underlying operating system inter-process communication mechanism and performance issues become important. The performance of inter process communication mechanisms depends upon the volume of traffic sent through the mechanism. Within window systems, the protocol for communication is defined at a higher level of abstraction than bit maps in order to reduce the volume of traffic. The X Window system [Scheifler 86] has commands which "draw circle" or "draw line" and graphical communication is handled at that level of

abstraction. The NeWS system [SUN 87] sends messages which carry PostScript programs [Adobe 85]. PostScript is a display formatting language described in more detail in Section 5.3.

This ability to communicate at higher levels of abstraction is also exploited to allow the client to change the interpretation of key or choice events. The use of PostScript allows the "downloading" of actual programs which can change any facet of the window system behavior.

The use of the operating system's inter-process communication mechanism means that the communication between the client and the process is asynchronous. Order of communications in one direction is maintained but the sequencing of messages is not. Each window could have a collection of clients and each client could have a variety of different windows being managed by the server.

4.5.2. Networking

The use of the operating systems inter-process communication mechanisms for communication between client and server allows the client and server to be distributed across a local area network. Figure 4.7 displays a network which exploits the distinction between clients and servers. A client resides on one workstation and can have a server which resides on a different physical workstation and manages a different physical terminal. The implications of that type of structure are still being explored for various client domains.

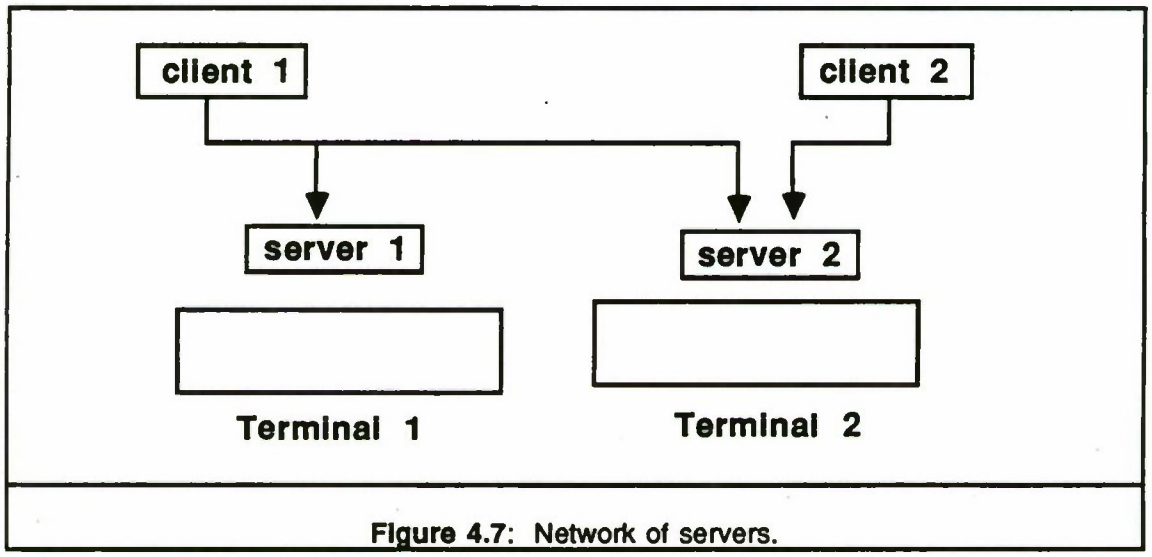


Figure 4.7: Network of servers.

4.6. Desirable Features of Window Systems

A number of the items discussed are important features in the evaluation of any window system. They are:

1. Does the system separate basic mechanisms for managing windows from the policies involved in the management. NeWS or the X window system, for example, support either tiling or overlapped windows. It is the

responsibility of the client to adopt a policy and the window system will provide the mechanisms.

2. Does the system provide one communication channel per client process. When this is so the client is guaranteed to receive events in the right order. If there were one communication channel per window then distinguishing the order of events across windows becomes difficult. Having one communication channel per client also avoids polling by the client on all of the channels to see if an event has arrived.

3. Does the system allow the definition of a hierarchy of windows. When using a direct manipulation interface, it is important to be able to handle object overlapping. Object overlapping is easily handled within a hierarchy of windows. Movement of the parent will move the entire object.

4. Does the system provide the client with offscreen bitmaps (or canvases with the same graphics operations as visible windows. If the client needs to distinguish between visible and obscured windows in order to perform basic operations then the interaction between the client and the window system becomes needlessly complex. Also, the offscreen bitmap acts as a cache for pixels and becomes a performance enhancement mechanism.

5. Does the system allow the clients visiblity into and use of non window management facilities. For example, communication between various clients is greatly simplified if the window systems communication facilities are available.

6. Does the system allow the clients control in the case of failures. For example, if the client requests an unavailable font then the window system should have a well defined, consist method of allowing the clients to determine strategy. This facility is important in the building of robust systems.

4.7. Rooms

A particularly interesting user interface which has been developed on top of a window system is Rooms by Card and Henderson [Card 87]. It is an example of how cognitive studies and information can be used to develop better user interface software.

The first step in the development of Rooms was to analyze the way in which people used windows. The data gathered showed that people used windows in groups. That is, there was one group of windows in which there was activity and that activity was localized in that one group and then activity was transferred to a second group of windows and activity was localized in that second group and so on. The pattern of activities supports the hypothesis that an end user performs one task at a time. The windows in which activity was localized were those windows which supported the particular task being performed.

The second step was the realization that the set of all existing windows could be collected into the groups within which activity was localized and that these groups

could be made the basis for a system. The metaphor of rooms in a building and the windows within each room was used as the basis for building a system.

In Rooms, the end user is provided with a collection of rooms in a building. Examples might be the mail room or the project meeting room. Within each room windows could be created or destroyed. A particular room is current at any point in time and within this room all of the windows are exposed. Rooms which are not current (in the metaphor, rooms in which there are no occupants) are represented as icons. Thus, when a user moves from one room to another (changes tasks), the windows in the room being exited become unavailable and the windows in the room being entered become available.

Each room is given a different background so the user can tell which room is currently active and an architectural plan of the building is kept available so that the user can determine how to navigate from one room to another.

There are a number of additional features to Rooms (window sharing and expanding upon the metaphor) but the heart of the system came from the realization that people used windows in a localized manner and that if the system supported this localization then windows would be used more efficiently. Pre and post studies showed that the typical user managed about three times as many windows using Rooms than using a normal window manager. Since users manage as many windows as they can comfortably handle, Rooms increased the number of windows with which a user is comfortable. Rooms is an outstanding example of the connection between understanding the cognitive machine of the end user and the requirements of software.

4.8. Introduction to Toolkits

The level of abstraction available from a window manager is really too low for convenient use by a client programmer. The client receives detailed knowledge of choice events (button up and button down are separate events, for example) and the ability to determine the location of the mouse cursor within a window. The client also specifies precisely the type of output to be placed within a window.

At a higher level of abstraction, the client programmer would have available a library of interaction objects. Each with its own geometry and behavior. Such things as command buttons, dials, sliders could be used to interact with the client at the level of "object selected" and "value set." These types of interactors are available in *toolkits* and are discussed extensively in the next section.

5. Toolkits

Tools for Implementing user Interfaces are becoming more available. Although they aim at the same goal, they are not all equivalent. The purpose of this chapter is to present a classification that organizes the space of existing tools into classes. Each class is characterized by the level of services it offers to the implementer. Tools for the construction of user interfaces range from the low-level toolkits to the more elaborate User Interface Management Systems (UIMS).

A brief taxonomy for user interface tools is presented in the first section. In Section 5.2, attention is focussed on toolkits per se. One important component of toolkits includes facilities for graphics. This topic is presented in the last section of the chapter. Sophisticated tools known as User Interface Management Systems are described in Chapter 6.

5.1. A Taxonomy of Tools for User Interface

As shown in Figure 5.1, tools for the development of user interfaces come in two categories: toolkits and User Interfaces Management Systems (UIMS).

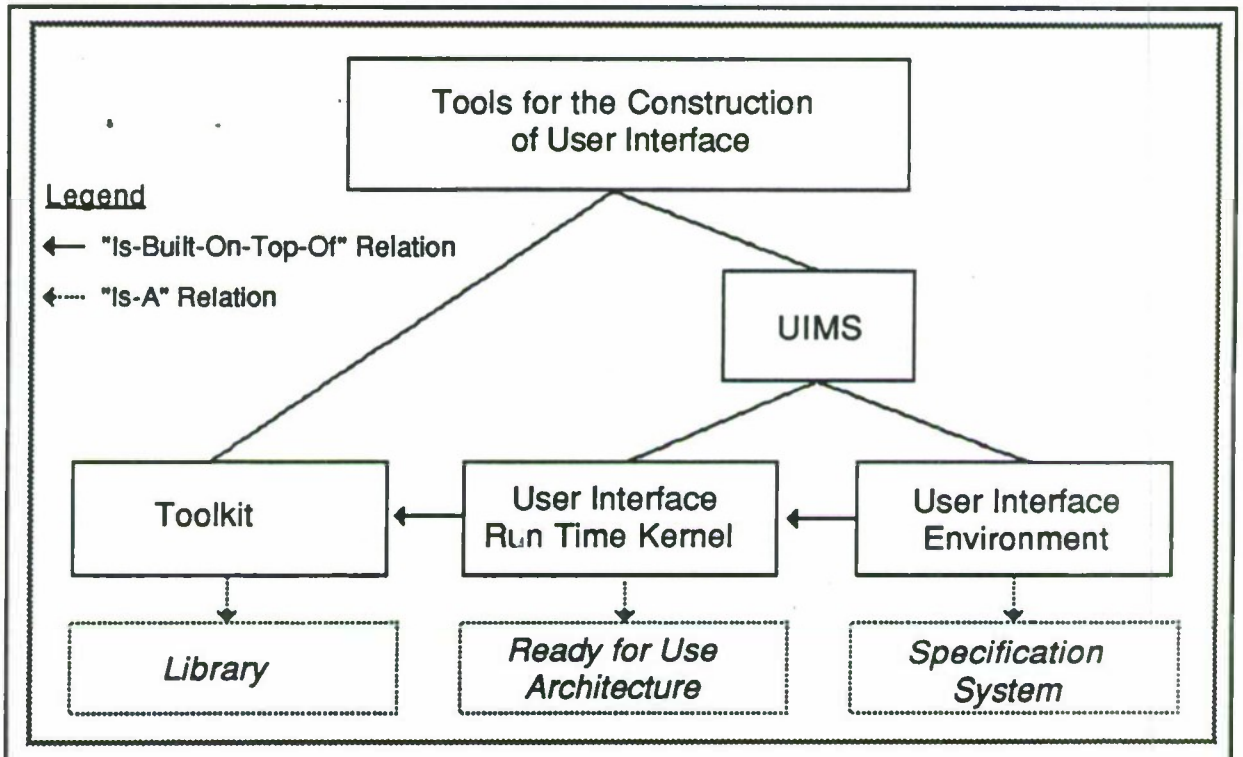


Figure 5.1: A taxonomy of tools for the construction of user interfaces.

A toolkit is a set of building blocks that the implementer assembles to manufacture a user interface. It provides the programmer with a wide range of functions from the low-level management of the workstation such as windowing, graphics, sound and text

editing, to the higher level of dialogue handling in the form of menus, buttons, control panels, etc.

User Interface Management Systems come in two forms: user interface run time kernels and user interface environments. A user interface run time kernel is a skeleton upon which the functional components of applications can be embedded. A user interface environment automatically generates a user interface from the specification provided by the designer and link the interface to the application. For doing so, it includes a run time kernel into which the application and the "compiled" specification are plugged.

In summary, toolkits provide the building blocks, run time kernels package the code that implements the foundation of an interactive system into a reusable and extensible skeleton, and a user interface environment automatically generates the specific aspects of a user interface from high-level specifications. When considering the ease of construction, the level of service increases from toolkits to user interface environments.

5.2. Toolkits

5.2.1. Overview: General Services

Figure 5.2 shows a classification of the types of services provided by any toolkit. These services can be organized in two categories: services related to the management of the workstation and services for the management of the dialogue.

Services for the management of the workstation define a virtual terminal as presented in Chapter 4. Abstractions vary from one toolkit to another, but they usually include:

- Foundations for graphics (e.g., offscreen bitmaps or canvas, viewports, windows).
- Primitive graphic entities (e.g., Icons, cursor shapes).
- Elements for text processing (fonts), and sound.
- Support for event handling.

Services for the management of the dialogue rely on the abstractions defined for the management of the workstation. They include:

- Elementary entities for dialogue handling such as buttons and scrollbars.
- Compound objects such as menus and forms.

In addition, recent toolkits such as X Toolkit, propose a model and a general mechanism for building special purpose dialogue objects.

For some toolkits, such as the Macintosh Toolbox [Rose 86], workstation management and dialogue management are gathered in a single library. For others, the distinction between the two levels of services is more explicit. For example, in the X-Windows environment, services for the management of the workstation are accessible through X-Lib whereas services for the management of the dialogue are gathered in X-Toolkit.

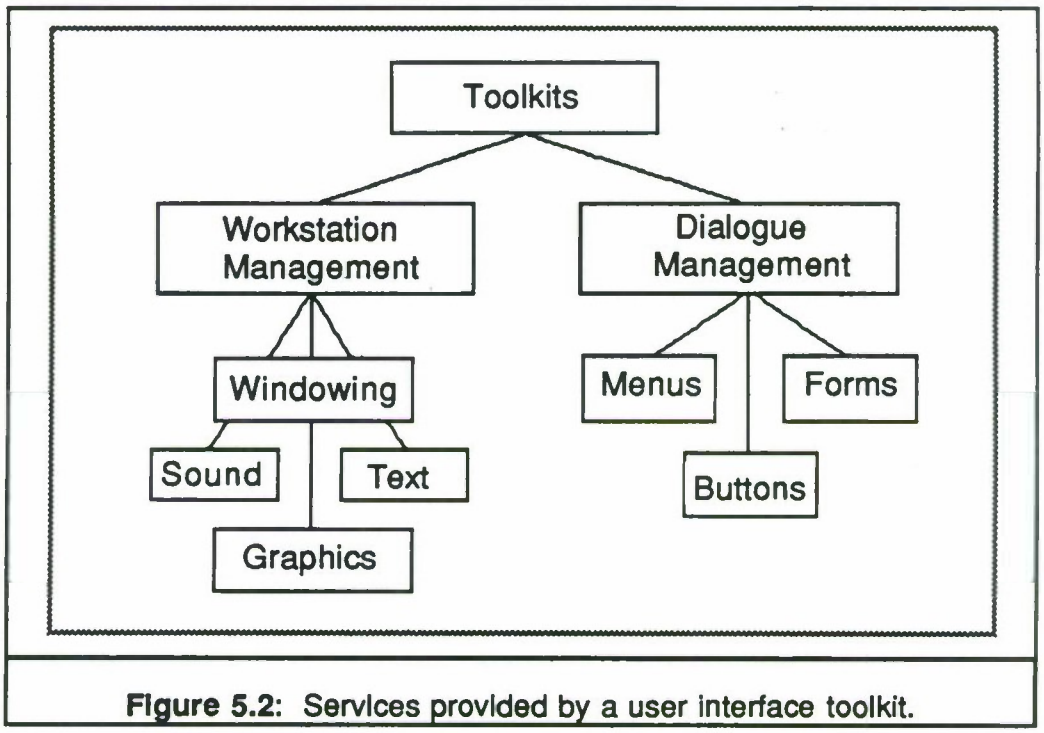


Figure 5.2: Services provided by a user interface toolkit.

5.2.2. Advantages and Drawbacks of Toolkits

5.2.2.1. Advantages

As for any library, a toolkit is a convenient support for portability and flexibility. Its last advantage is specific to the domain to which it applies by defining a consistent style of interaction.

1. Portability. Software portability is one of these practical problems that computer scientists face continuously. Knowing that the user interface part of an interactive system can represent up to 80% of the code, the portability of user interfaces deserves special attention. Toolkits offer a convenient and natural way for defining levels of portability.

2. Flexibility. Software flexibility covers issues about diversity and extensibility. Diversity is concerned with the availability of various levels of abstractions. With user interface toolkits, the programmer has the choice between the low-level services that allow him to control the workstation at a very fine grained detail and high-level services that provide him with ready for use local dialogues. Extensibility is the ability to add new features. As mentioned earlier, recent toolkits provide the programmer with a mechanism for building new interaction techniques. Other toolkits, in particular those integrated to an object-oriented

environment, encourage software reutilization through the subclassing mechanism.

3. Consistent style of interaction. Toolkits include a variety of interaction techniques that can be reused from one application to another. As a result, they define a style of interaction with which the user can progressively become familiar. In addition, the behavior of the interaction techniques has been determined in accordance with ergonomics principles. For example, in order to facilitate the evaluation stage, a button displays itself in reverse video as it is visited by the mouse.

The ability to determine the arrangement of the building blocks allows the implementer to fully control the behavior of a user interface. Unfortunately, this freedom has its counterparts.

5.2.2.2. Drawbacks

Toolkits do not embed any software architecture; they are hard to use and they lead to duplication of efforts.

1. Wrong Software Architecture. A library does not embed an architecture. In particular, user interface toolkits do not enforce the modular distinction between the application and the user interface. As a result, toolkits may lead to suspicious software architectures where the expression of the user interface is mixed with the expression of domain dependent functions. Mixing the two aspects impedes the maintenance of the interactive system and does not make it possible to iteratively adjust the user interface.

2. Long Learning Phase. As Figure 5.3 demonstrates, a toolkit is a big bag of functions. Finding the right arrangement may be a tremendous technical barrier specially for the first time developer.

3. Duplication of Efforts. Making the glue must be carried out for each interactive system. It is not surprising then that a strong interest has recently emerged for run time kernels that provide implementers with reusable code organized in a ready for use architecture. This facility will be further described in Chapter 6.

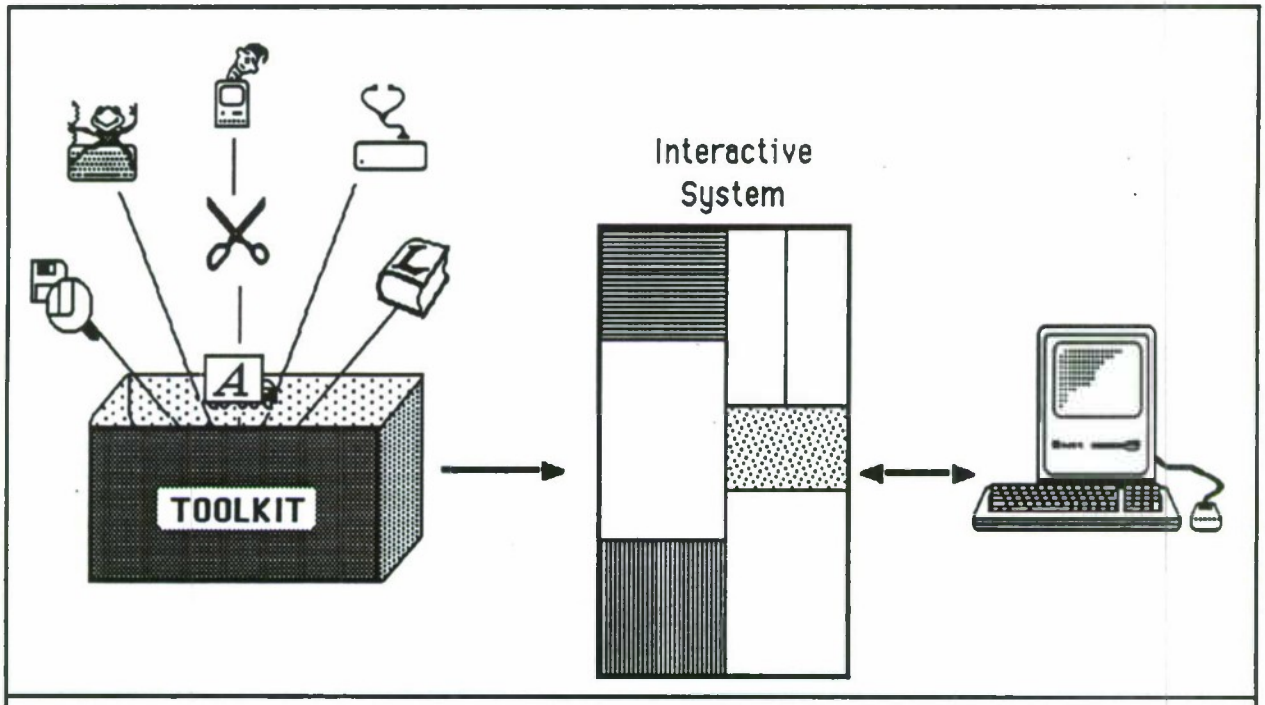


Figure 5.3: Toolkits provide the implementer with a set of building blocks to be glued together.

5.2.3. Comparative Analysis

Toolkits differ mainly in the control strategy they embed, the ability to allow the programmer and the user to overload and customize presentation policies, and facilities for implementing direct manipulation interfaces. This issues are successively developed in the next paragraphs.

5.2.3.1. Control Strategy

Protocols for acquiring and processing events have a strong impact on the control structure of a system. With regard to user interface toolkits, there are two types of protocols whether or not the control strategy is embedded in interaction techniques.

When the control strategy is embedded, the interaction techniques have a mechanism to process events. This mechanism is automatically activated when an event is of interest to the technique. (A technique can express interests for classes of events at any time). When the technique has completed processing an event, it automatically calls a procedure provided by the client program. This procedure performs some domain dependent computation. If no callback procedure has been specified for the event class, there is no further processing. It means that this event class has no domain dependent meaning. X-Toolkit widgets and NeWS interactive objects are built according to this policy.

When the control strategy is not embedded in interaction techniques, the processing sequence has to be specified by hand. The programmer needs to explicitly ask each possible interaction techniques whether it is concerned by the event. If so, the programmer chooses one of the possible methods attached to the techniques. The

technique has no event handler. It has a collection of methods that can be invoked. The technique is not an agent endowed with capabilities for decision making. It is a passive server. The Macintosh Toolbox is based on a non embedded control strategy.

To summarize, the embedded control strategy automatically performs the sequence of actions for processing events and client programs are called for complementary processing. At the opposite, when the control strategy is not embedded, the programmer is in charge of gluing the pieces of processing together.

5.2.3.2. Overloading and Customizing Interaction Techniques

A consistent style of interaction is a desirable feature. However, the style defined by a toolkit cannot be expected to be satisfactory for every situation. In some circumstances standard behavior needs to be adjusted. The adjustment can be performed either by the programmer or by the user.

Programmers may desire to modify the visible behavior of an interactive object or the internal functional behavior. Toolkits based on the object-oriented paradigm such as ones in the Smalltalk-80 [Goldberg 84] or Loops [Bobrow 83] environments encourage such overloading: the programmer defines a new subclass and overloads the inherited methods with his special purpose code. Toolkits such as the Macintosh Toolbox, although they claim to be designed according to the object-oriented paradigm, make the modification much harder, hard enough to be discouraging!

Users may want to customize a user interface without getting involved in a programming task. The type of customization that is currently feasible without programming is concerned with the lexical level only. For doing so, a toolkit must provide an external permanent representation for interaction techniques. External, means that the description of the interaction technique is not wired in the code of the user interface. Permanent, means that the existence of the representation is not tied up to the execution of the interactive system. Files provide a convenient way for maintaining permanent data. Finally, the external representation can serve as input data to an editor which allows the user to interactively customize the lexical aspects of the interaction techniques. The notion of resource developed for the Macintosh Toolbox is an excellent illustration of how lexical customization can be performed by any user.

5.2.3.3. Facilities for Implementing Direct Manipulation Interfaces

User interfaces based on the direct manipulation metaphor are very demanding on the software side. In particular, an object may, as a whole, be constrained to follow the movements of the mouse and, as a part, be locally edited in real time.

Mouse tracking requires a loop of three software actions: erase the object from its previous location, repair the surface that has been damaged, and draw the object at the new location. Current toolkits do not provide much support for satisfying these requirements. The Macintosh Toolbox offers the notion of region that the client program can drag around as long as the user holds the mouse button down (cf primitive DragGreyRgn). However, this local facility, although very convenient, is not a general mechanism to deal with overlapping objects. X Windows with its recursive notion of overlapping windows offers an attractive foundation for implementing overlapping objects.

Editing part of an object is a second heavy requirement on software programming. Objects are usually compound entities. Sometimes, they are treated as wholes (as in mouse tracking) and sometimes as parts (as in editing tasks). Graphics tools available in user interface toolkits either do not have any facilities for encapsulation or they have encapsulation facilities which hide access to the parts. In the first case, there is no way to consider the object as a whole. In the second case, there is no way to edit part of the object. For example, pictures and regions of the Macintosh Toolbox, and GKS segments are like graphics macrocommands. The client program can execute them with different parameters involving location, rotation and scaling. Pictures, regions and segments are mechanisms for encapsulation. They allow for the definition of a graphics object from elementary graphics primitives. However, if the client program needs to modify a line segment of the object as the user moves the mouse, the picture, the region and the segment do not allow this. The picture, region and segment must be destroyed and rebuilt with the new line segment! The following paragraph describes graphics tools that are more appropriate for interactively editing graphics objects.

5.3. Graphics Tools for Abstract Imaging

Information layout can be viewed as a sequence of transformations from internal domain dependent data structures to actual images. Information acquisition from a selected point in an actual image to some internal data structure is the reverse sequence of transformations. This subsection presents two general techniques that automatically perform these two way transformations. The first category focuses attention on structural relationships between the components of an image. The second one is based on a general constraint problem solver approach. Before describing these techniques, we need to briefly review low-level graphics tools.

5.3.1. Low-level Graphics Tools

Low-level graphics tools such as CGI [ISO 86b] define a graphics machine for drawing lines, circles etc. in a graphics space coordinate. Other tools such as PostScript [Adobe 85], QuickDraw [Rose 86] and GKS [ISO 85] include a simple encapsulation mechanism. They respectively propose the notions of path, region/picture and segment. Although encapsulation is a convenient way for grouping logically connected information, it is not adequate for interactively editing parts of graphics compound objects. PostScript, however, deserves additional comments.

PostScript is a powerful programming language that has the ability to describe the appearance of any type of information on a rendition surface (paper or screen). Its power is Turing equivalent; the syntax incorporates a postfix notation and the data model includes, like LISP, the ability to treat programs as data. PostScript imaging model is very general and very simple. Figure 5.4 illustrates the model. Imaging is based on a stencil/paint model. A stencil is an outline specified by an infinitely thin boundary that is piecewise composed of spline curves. Paint is some pure color or texture or even an image which is be dropped on the drawing surface through the stencil. PostScript has been extended to serve as the programming interface for NeWS: client programs are not limited to a predetermined set of requests but they can download PostScript programs to the NeWS server.

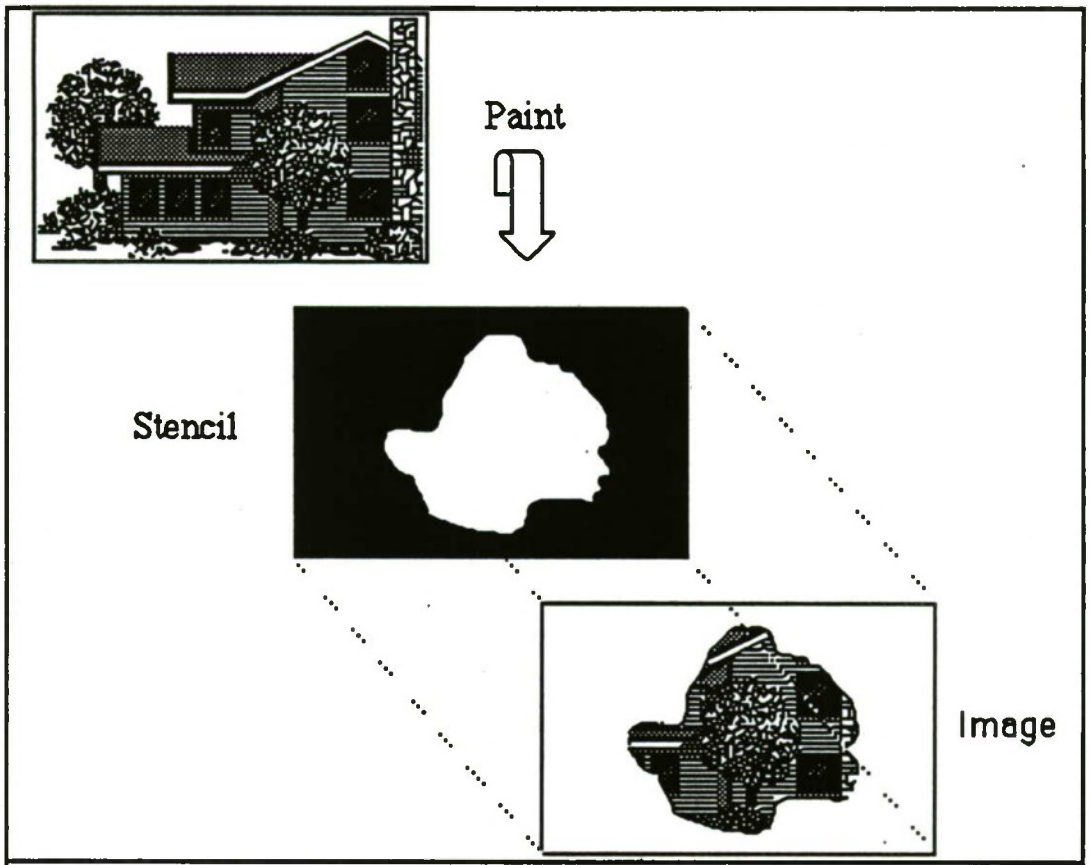


Figure 5.4: The PostScript imaging model.

Figure 5.5 gives an overview of the level of abstractions of graphics tools.

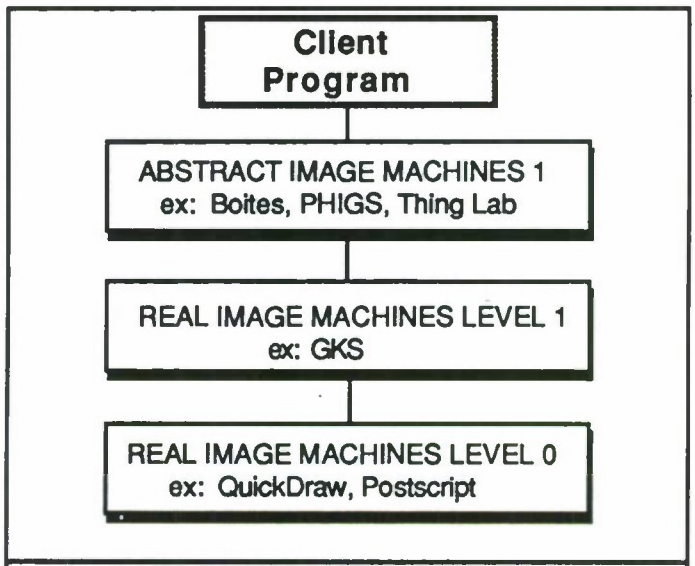


Figure 5.5: How graphics tools relate to each other with regard to their level of abstraction.

5.3.2. Abstract Imaging and Structural Relationship

As described in Chapter 3, an abstract image is an intermediary data structure between structures maintained by the application program and the actual image on a rendition surface. It shortens the distance between the representation convenient for the application program and the representation required by windowing systems. Its purpose is to express logical relationships maintained in the application data structures into graphic relations. The goal is not to express all of the logical relationships but the relationships that help the user perform the execution and the evaluation stages. One important class of relations is the structural relationship. A number of tools based on the notion of box and the graphics ISO standard PHIGS propose abstract imaging around the notion of structure.

5.3.2.1. Box-Based Abstract Imaging

The notion of box has first been used for TEX [Knuth 79] for output rendition only. Since then, the notion of box has been extended by a number of tools [Mikelsons 81, Coutaz 85a, Coutaz 85b, Alhers 86, Quint 87] to consider inputs as well.

The box as described in [Coutaz 85a and Coutaz 85b] is a tree-like structure. A tree facilitates the definition of an inheritance and a synthesis mechanisms for computing attributes. Attributes decorate nodes to express spatial relations (such as alignment and indentation), visual effects (such as highlighting and coloring), polymorphism (such as elision), and links to application dependent data structures. Leaves contain displayable application dependent information. They are recipients. They do not have any semantic knowledge about their content but its type (e.g. image, text). As a recipient, a leaf wraps an imaginary rectangle around the information. Nodes are

compound boxes. A compound box is the result of a formatting composition from subtrees.

Figure 5.6 shows one possible tree of boxes that corresponds to an "if statement" maintained by a syntactic editor.

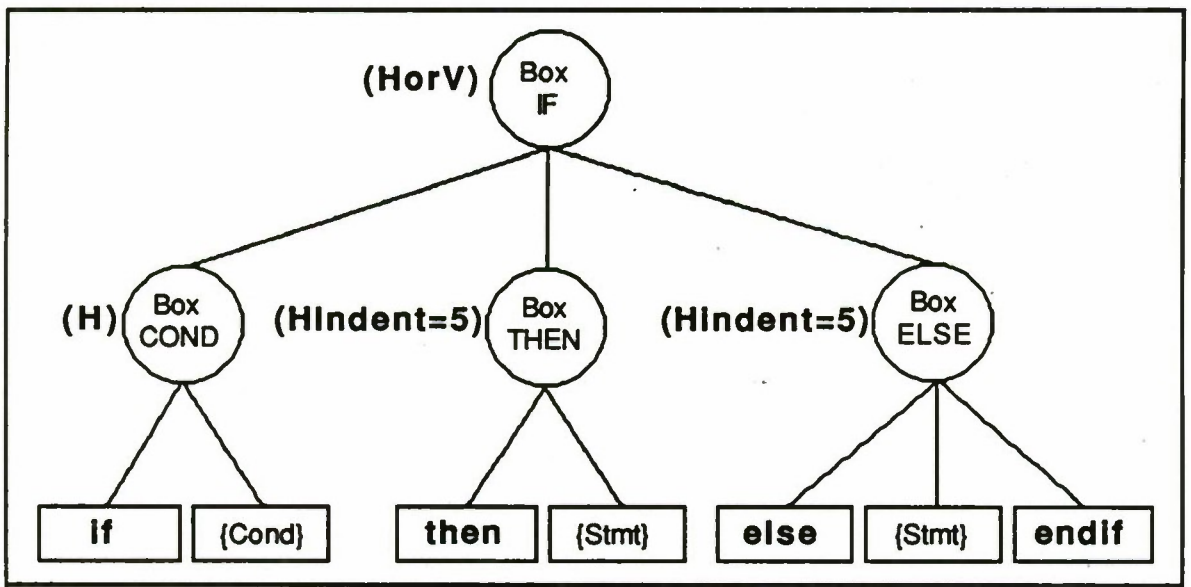


Figure 5.6: A tree of boxes as an abstract Image for an "IF statement".

The formatting attributes HorV first tries to concatenate the subtree horizontally. If the resulting rectangle is too wide to fit the available width of the rendition surface, a vertical composition is applied automatically. The attribute H concatenates the subtrees horizontally. HInd specifies the value of the horizontal indentation if one has to be performed.

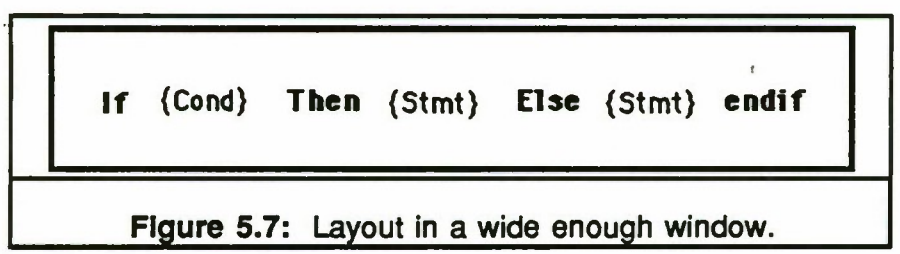


Figure 5.7: Layout in a wide enough window.

The interpretation of the tree will generate the actual images shown in Figures 5.7 and 5.8 depending on the effective width of the output window. Note that when the user resizes the window, the new formatting is automatically handled by the abstract Image interpreter. The application is not bothered by syntactic user actions that are irrelevant to its expertise.

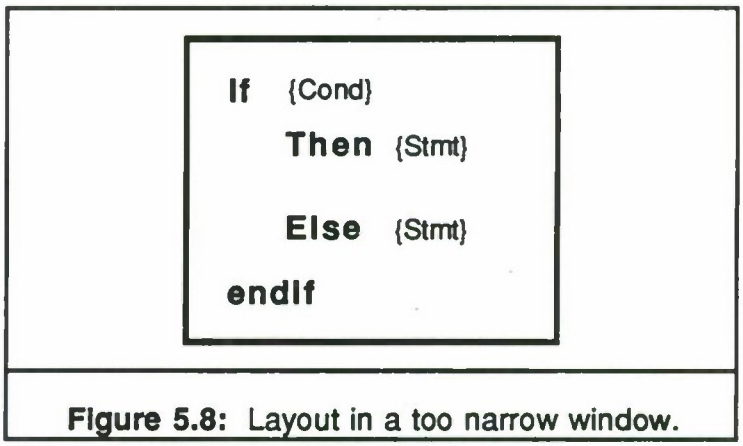


Figure 5.8: Layout in a too narrow window.

5.3.2.2. PHIGS

PHIGS [ISO 86a] is a standard for graphics which takes GKS as a point of departure. However, the static notion of segment has been replaced by the editable notion of structure. Figure 5.9 shows an example of a structure definition. The interpretation of the request POST_STRUCTURE(A) executes the definition of A. The definition of A is comprised of graphics elements included between the requests OPEN_STRUCTURE(A) and CLOSE_STRUCTURE. The element EXECUTE_STRUCTURE behaves just like a procedure call: it saves the current context, deviates to a new context and comes back to the calling context. EXECUTE_STRUCTURE(B) saves the current graphics context about A, interprets the definition of B and, once B has been made part of A, returns to the execution of A. For inputs, PHIGS uses an extension of the GKS notion of logical units to take into account the structural organization. In particular, a PICK returns a path which uniquely denotes the selected element.

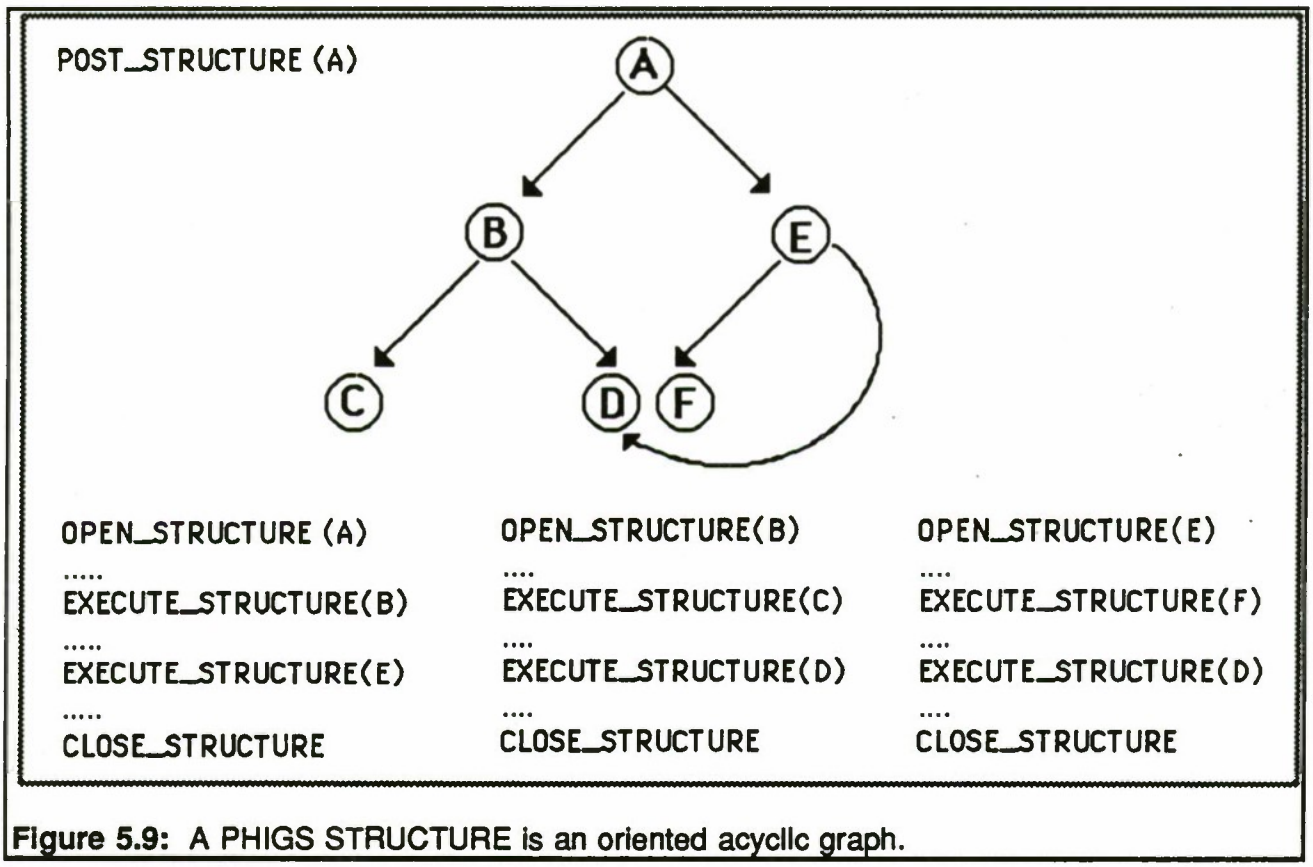


Figure 5.9: A PHIGS STRUCTURE is an oriented acyclic graph.

In contrast to GKS segments, PHIGS structures can be dynamically modified. The model for modification is inspired from line text editors. Figure 5.10 shows an example of a structure edition. As for text editors, you first need to open the recipient: OPEN_STRUCTURE(MYHOUSE) opens the structure MYHOUSE. By doing so, the interpreter places the insertion point at the end of the structure definition and sets itself in input mode. This means that subsequent graphics elements will be automatically added at the end of the current structure. If the client program needs to delete the window element, then a DELETE_ELEMENT(MYWINDOW) will do the job. The LABEL (MYWINDOW) is a symbolic way of denoting a graphics element, just like a line number designates text lines in line based text editors. Similarly, if one wants to replace the definition of the door, then the insertion point can be set at the appropriate point in the structure definition and the replace mode will substitute old graphics elements by new ones.

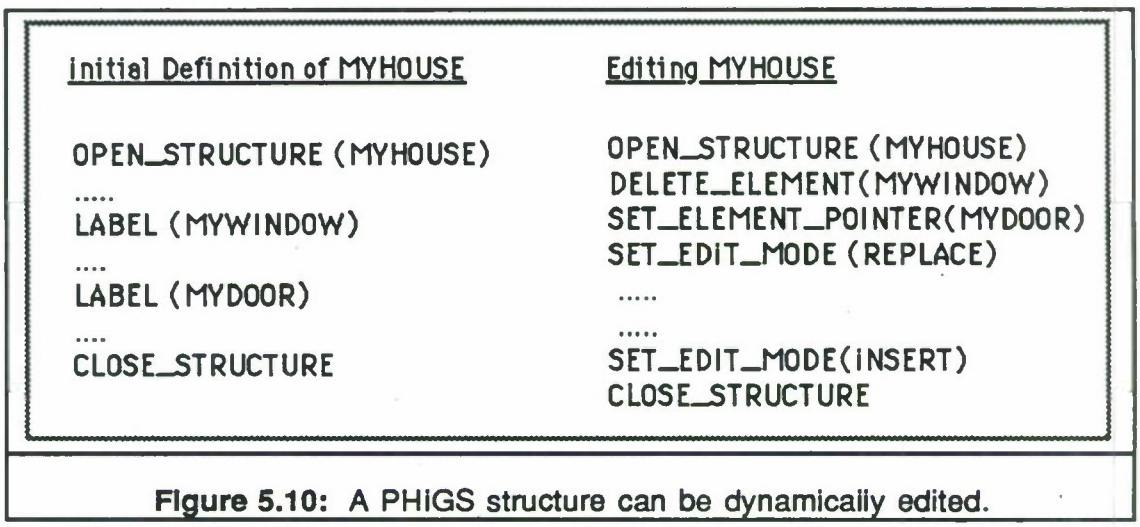


Figure 5.10: A PHIGS structure can be dynamically edited.

5.3.3. Constraint-Based Imaging

A constraint describes a relation which must always be satisfied. The set of relations maintained in the abstract image machines presented in Paragraph 5.3.2 is limited in scope. More general mechanisms for expressing any type of graphics constraints need to be developed. ThingLab [Borning 86], although its goal is not abstract imaging, is an interesting illustration of a graphics constraint solver.

ThingLab is an interactive environment built on top of Smalltalk-80. It allows a user to specify constraints between graphics objects. At the opposite of the box mechanism, these constraints are not restricted to a predetermined set. A ThingLab constraint is comprised of a predicate and one or several methods. The predicate is an algebraic expression which is used for constraint checking. The methods modify the entities referenced in the predicate in order to guarantee visual consistency. The power of ThingLab is that these methods are automatically generated from the specification of the predicate. Figure 5.11 shows an example.

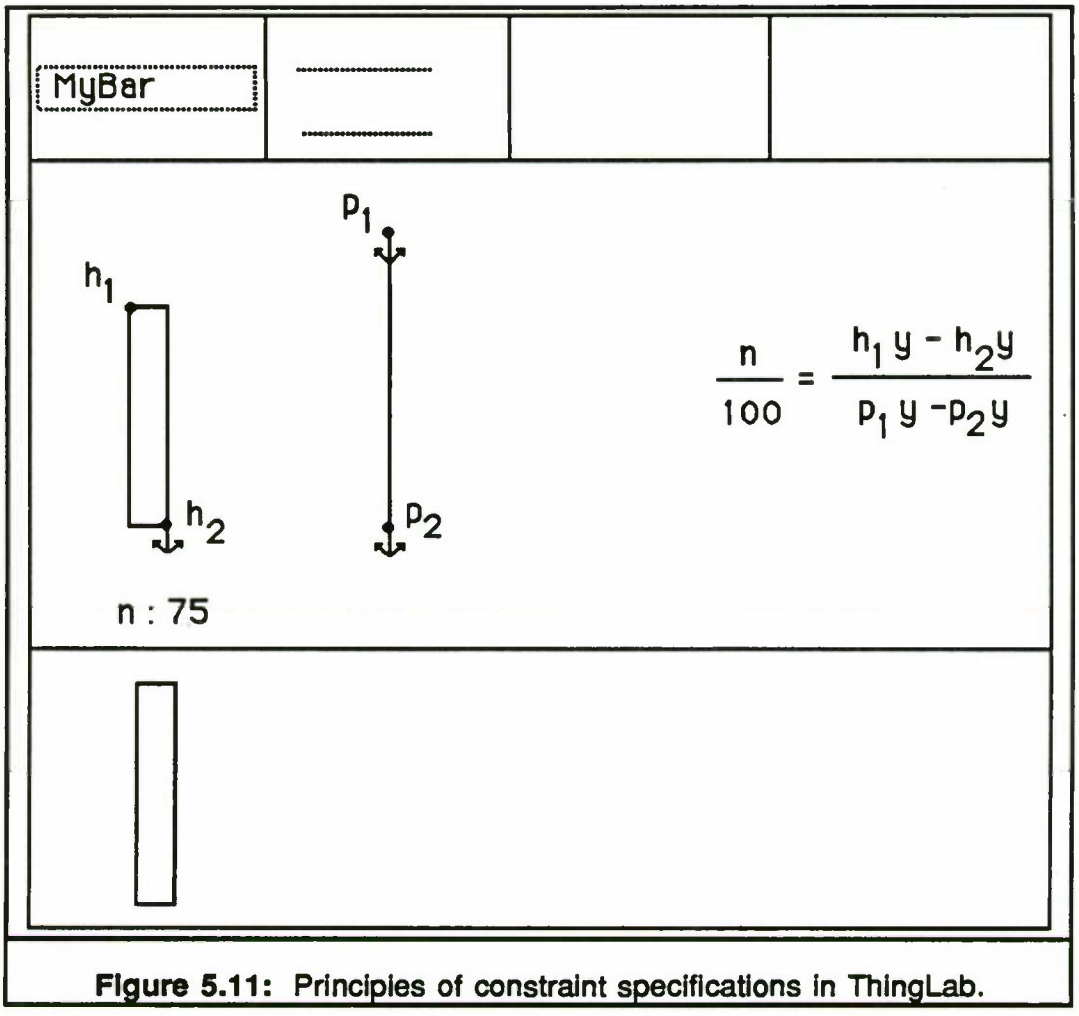


Figure 5.11: Principles of constraint specifications in ThingLab.

The bottom window contains the object MyBar as it will appear at runtime. The upper window gathers the usual Smalltalk browser menus which allow the user to define an algebraic expression, identify the constants, the variables, and indicate which class is reused to build the new object (currently, the rectangle class is appropriate to construct MyBar). The middle window is the workshop. The goal is to define a vertical bar to represent an integer n comprised between 0 and 100. The algebraic expression defines the height of the rectangle where: h1 and h2 are respectively the top left and bottom right corners of the rectangle; p1 and p2 are two constant points such that the length of the segment [p1p2] determines the height of the rectangle when n is 100; h_1y and h_2y denote the vertical coordinates of H1 and h2.

ThingLab has served as a basis for the implementation of more specialized environments: Animus [Duisberg 86], which introduces the notion of time, and the Filter Browser [Ege 87] for the specification of user interfaces.

6. User Interface Management Systems (UIMS)

6.1. User Interface Runtime Kernels

6.1.1. Introduction

Toolkits provide components with which it is possible to construct a user interface. Each component is specific to a particular information presentation or acquisition task. A complete interface, however, must contain multiple components which act together to convey information to and from the functional portion of the interactive system.

A user interface runtime kernel is a skeleton or a packaging of the tools in a toolkit to provide a collection and a sequencing mechanism for the tools and a communication mechanism for information to and from the functional portion. The issues involved in the runtime kernel are:

1. The software structure used in the runtime kernel. In particular, the architectural model underlying the software and the interface between the particular components of the architectural model.

2. Threads of control.

3. The model used to describe the interactions between the end user and the functional portion. This is usually called the dialogue model.

4. The management of multiple views of the same application data instance.

5. Feedback issues.

These issues are discussed in the sections that follow.

6.1.2. Software Structure

There is general agreement that a complete interactive application can be partitioned into three components [Pfaff 85]. These three components are the functional core of the application, the user interface runtime kernel and the lower level presentation layer. Each component can be implemented using whatever tools are available. In Chapters 4 and 5, the presentation layer has been discussed in terms of window systems and toolboxes. In this section some of the structural issues associated with the runtime kernel are discussed. In particular, a method for dealing with the interfaces between the layers based on the Serpent UIMS [Bass 88] and a method for using an object-oriented decomposition of the runtime structure based on the PAC model [Coutaz 87a, 87b] are discussed.

Figure 6.1 gives a high-level view of the components of an interactive application. The application component consists of the functional core and a communication portion with the user interface. The objects in this communication portion are at the level of abstraction of the application and have no presentation components. The user

interface has two portions: the presentation components and the dialogue controller. The objects in the presentation component are presentation objects and have no application knowledge within them. The behavior of these objects must convey application semantics but the objects themselves have no application knowledge. The dialogue controller performs the mapping between the application objects and the presentation objects. Application domain knowledge can be embedded into the dialogue controller to perform the mappings or can be restricted to the functional core. These decisions depend upon the particular circumstances of the application.

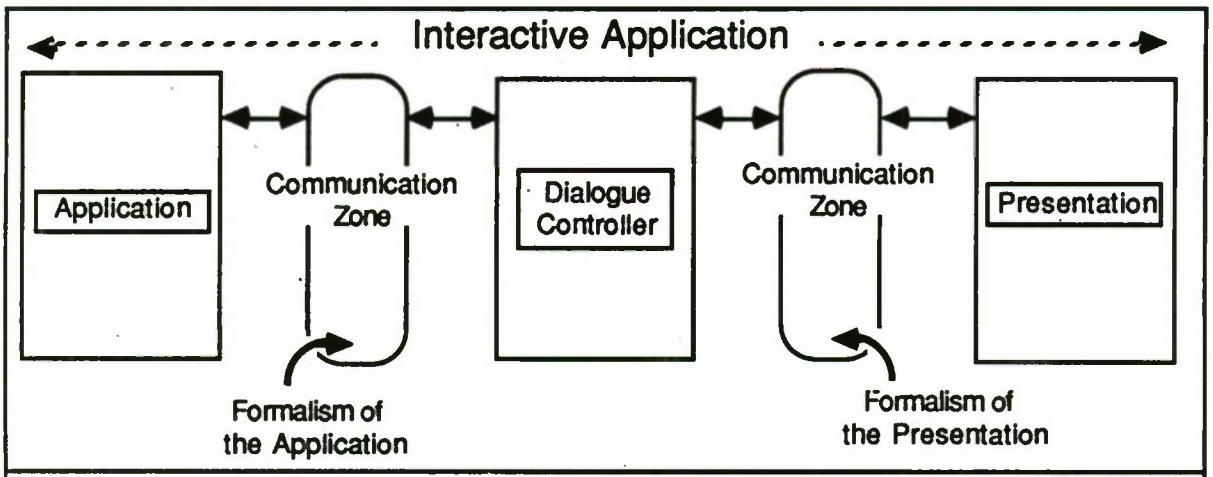


Figure 6.1: High-level view of the components of an interactive application.

Note that the mapping between the application objects and the presentation objects is bidirectional. End user actions will both modify the application objects and provide commands to the application core to perform its functions. Also, the mapping is not necessarily one to one. Suppose the display shows a fluid boiling. The application has one object which represents temperature and another which represents pressure. The boiling point depends upon both. The dialogue controller must combine the two application objects into a single presentation object. This is an example of a situation where the dialogue controller has application domain knowledge.

6.1.3. Serpent Component Interface Management

Serpent is an example of such a runtime kernel and will be used to explain the concepts in more detail. An application using Serpent has explicitly three components. These are: the application functional core, the runtime kernel and the presentation level. The presentation level is composed of an X toolkit component and other components which use different technologies for input and output (e.g. video output and gesturing input). Serpent is designed to allow for easy integration of additional interaction mechanisms and explicit separation between the application functional core and the runtime kernel. The integration of additional interaction mechanisms is accomplished by having an explicit separation between the presentation layer and the dialogue manager. The interface between the layers allows for different presentation layers with only a modification of the dialogue manager and no modification of the application.

The separation between the components is accomplished by providing an explicit interface description. On one side of the interface is the Serpent runtime kernel. On the other is either the functional core of the application or the presentation layer. Figure 6.2 displays this structure.

The application and the presentation layer view the Serpent runtime kernel as an active data base manager. The application views Serpent as a manager of data of which the end user might be interested and the presentation layer views Serpent as the manager of data which control their presentation and interactions. In either case, there is an explicit specification of the data which is to go through the Serpent runtime kernel. This specification takes the form of a schema which is similar in form to a schema for a traditional data base system.

Whenever the application modifies a data item in the data base managed by Serpent then the runtime kernel of Serpent manages all of the implications of that. When the end user performs an action which affects a data item in the data base which Serpent manages for the application then the application is informed of the change.

The schema which defines the form of the data to pass over the interface is processed prior to Serpent runtime. The processor produces a C header file (or Ada package) for the application to include. This guarantees that both sides of the interface have the same data description and, consequently, helps insure the integrity of the data which crosses the interface.

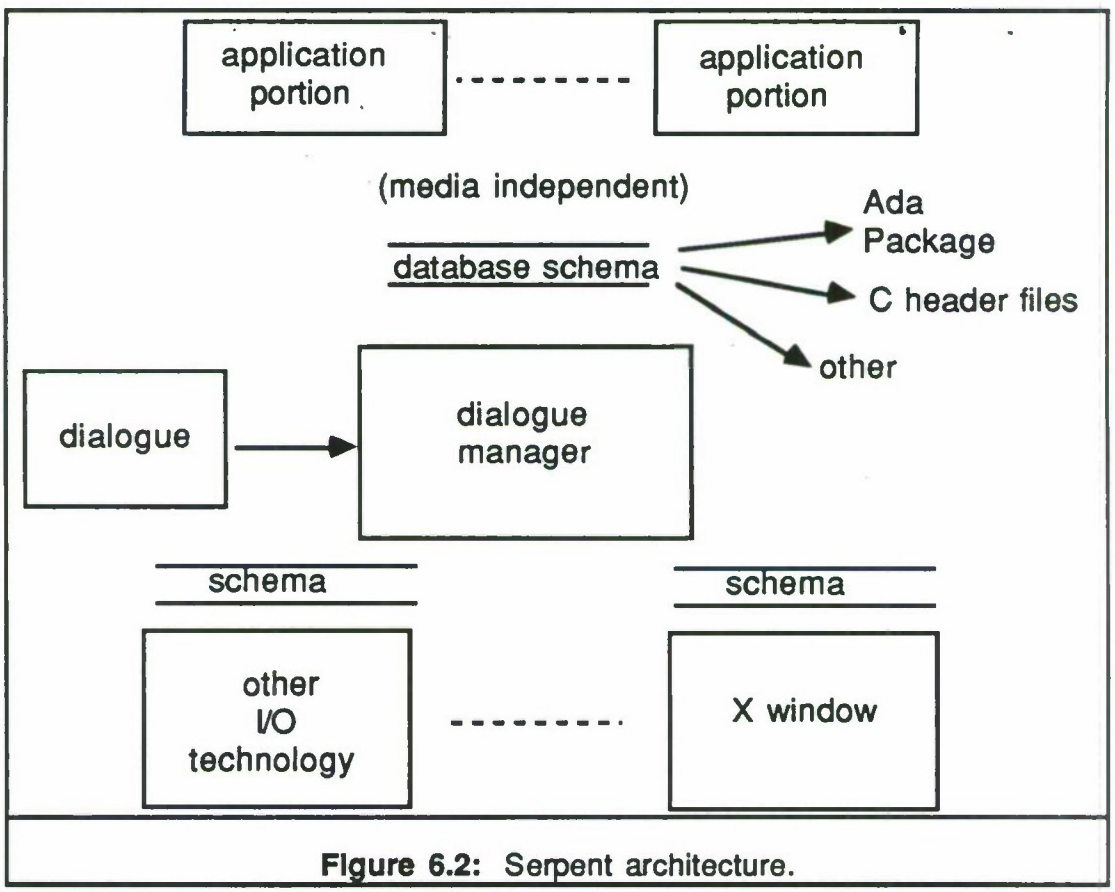


Figure 6.2: Serpent architecture.

The use of a schema to define the data that Serpent manages allows Serpent to be reusable. Data of arbitrary complexity can be described in terms of the schema description used in the interface and, consequently, additional interaction mechanisms can be added and arbitrary applications can use Serpent.

6.1.4. Threads of Control

One motivation for the Rooms system (Section 4.7) is the end user's desire to move from one task to another, whether the current task is completed. The dialogue controller must be able to maintain the context for the interrupted task and restore it when that task is to be resumed. This is one example of having *multiple threads of control* within a dialogue. Another example is the simultaneous use of multiple input and output devices. Some types of interaction require two handed input utilizing different devices [Buxton 86a]. If the devices are not integrated at the presentation level then the dialogue manager must simultaneously process the input from both devices, coordinate it and determine the mapping into desired application actions. In the Macintosh toolkit, for example, this type of activity must be performed in the top level controller and cannot be pushed into the presentation level.
In either case, the requirements imposed on the dialogue manager by both the end user task switching and the multiple simultaneous devices mean that the dialogue manager must support parallelism.

6.1.5. The Model Used to Describe User Interactions

A number of different models have been used to describe (and hence to specify) the user interactions. These models are:

1. Formal grammar models, in particular BNF
2. Finite state machines, usually augmented
3. Production or event models
4. Object-oriented models

Any implementation of these models has two portions. First is a language for describing interactions in terms of the model. A program in this language becomes a specification of the behavior of the runtime kernel. The second portion of the model is the runtime interpretation of the specification. An implementation decision is whether the specification language is compiled into a lower level description or is directly interpreted.

6.1.5.1. Formal Grammar Models

An early system, SYNGRAPH [Olsen 83], used BNF to specify the user interactions. Each non terminal in the BNF had an associated action routine which describes the presentation and the actions associated with the presentation. A legal interaction is one which can be parsed through the BNF. BNF, by its nature, has an explicit legal sequence of ordering of events. This imposes a particular style upon the interfaces specified using BNF. For example, suppose different parameter orderings are allowed.

Different BNF rules must be used to specify each ordering. Therefore, in order to allow the end user to choose an ordering at runtime multiple sets of BNF rules must be specified.

Furthermore, since all actions in BNF must be explicitly stated, allowing a user to change the current task in the middle (which, as has been described in Section 6.3 is a desirable feature), specifying complete interactions using BNF is a formidable chore.

6.1.5.2. Transition Networks

An alternative to BNF as a specification model is to use a finite state machine. The finite state machine is typically augmented to allow a richer description mechanism than finite state automata. USE [Wasserman 85] is an example of such a system. Finite state machines suffer from the same sequencing problems as BNF. An additional problem that both specification techniques suffer from is lack of model support for levels of abstraction.

The specification of a selection of an object (cursor over object, button click) is one level of abstraction, the specification of the ordering of parameters to a command is a higher level. A transition network does not distinguish between these levels of abstraction and, consequently, a specification using a transition network becomes difficult to code and decipher.

Some extensions to transition networks allow the nesting of transitions in an attempt to support the different levels of abstraction [Kieras 85, Harel 87].

6.1.5.3. Production Model

Production models are collections of rules of the form *if "firing rule" then "action"*. Productions are data driven in the sense that the rules are fired when the firing rules are satisfied and no particular sequencing constraints are placed on the firing rules. Production rules [Garrett 82, Hill 87a, Hill 87b, Brownston 85] have been used recently to attempt to specify the parallelism that end users seem to require. The CLG [Moran 81] is also a use of the concepts of production models for describing the interaction level although not explicitly discussed.

The Serpent model for dialogue uses "view controllers" to specify the mapping between the application objects and the presentation objects. Each view controller has a creation condition which corresponds to the firing rule. The creation condition is a condition on the application objects or on local objects. Local objects are maintained for dialogue control purposes only and are not visible to either the application or the presentation. Each view controller controls a collection of presentation objects. The methods of these presentation objects perform the reverse mapping from the presentation layer to the application. View controllers can be nested and the lower levels inherit the application objects which created the parent levels. The use of production rules solves the explicit ordering problems associated with transition networks and BNF grammars. On the other hand, there is still no model support for levels of abstraction. The support for levels of abstraction comes from the structural ideas of PAC or the nested objects used in the production model of Serpent.

Systems based on production rules suffer from several problems. Since control is not explicitly transferred within the specification of the dialogue, the system must monitor a large data space in order to decide which rules to fire. This monitoring of a large data

space may lead to performance problems. The appearance of more efficient production systems [Forgy 84] has reduced the magnitude of this problem. Preliminary indications are that performance within Serpent (which uses OPS83) is driven by the performance of the presentation layer and not by the production manager.

A second problem associated with the use of a production rule model is, precisely, the lack of explicit transfer of control. Programmers are taught to think of algorithms sequentially and the data driven nature of production models requires a heavily parallel method of thinking. This is a problem that can be overcome with training and if production rule systems prove to be suitably useful, then programmers will be taught earlier to think in terms of parallel solutions to problems.

6.1.5.4. Object-Oriented Model

A different approach to the specification of the mapping from application objects to presentation objects is to use an object-oriented approach. This approach underlies the PAC model [Coutaz 87b].

In the PAC model, an interactive application is comprised of three parts: Presentation, Abstraction and Control.

The Presentation defines the concrete syntax of the application, i.e., the input and output behaviour of the application as perceived by the user. The Abstraction part corresponds to the semantics of the application. It implements the functions that the application is able to perform. The Control part maintains the mapping and the consistency between the abstract entities involved in the interaction and implemented in the Abstract part, and their presentation to the user. It embodies the boundary between semantics and syntax.

For example, the application "Clock" implements and involves two abstract entities in the dialogue: the data structure "Time" and the function "SetTime". "Time" may be presented as a digital or a dial clock, SetTime may be explicitly presented as a button or implicitly presented through the direct manipulation of the needles of the dial clock. The job of the Control part is to invoke SetTime on specific user's actions and provoke the update of the dial clock when the application (i.e the Abstract part) makes a request.

The Presentation of an application is implemented with a set of entities, called interactive objects, specialized for man-machine communication. As with applications, an interactive object is organized according to the PAC model. Consider for example the pie chart shown in the Figure 6.3.

1. The Presentation is comprised of:
   - for output—a circular shape and a color for each piece of the pie.
   - for input—the mouse actions that the user can perform to interactively change the relative size of the pieces.
2. The Abstraction is comprised of an integer value within the range of two integer limits.

3. The Control maintains the consistency between the Presentation and the Abstraction. For example, if the user modifies the size of one piece, Control provokes the update of the integer value. Conversely, if the application or another Interactive object modifies the value of the Integer, the size of the pieces is automatically adjusted.

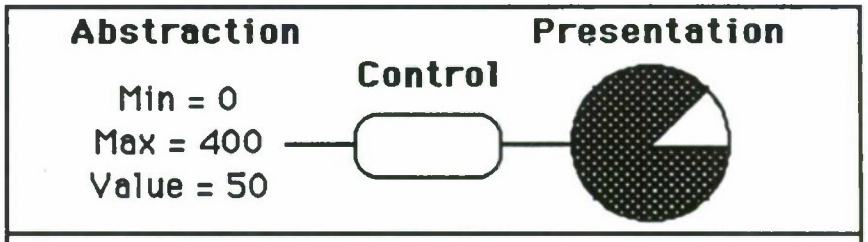


Figure 6.3: An elementary PAC interactive object.

Compound objects can be built from elementary interactive objects. They also adhere to the PAC model. Consider, for example, the super pie chart shown in the Figure 6.4. It is made from two elementary objects: the pie chart described above and a numerical string which shows the current abstract value of the pie chart. If Control C receives a message notifying him of the modification of the abstract value, it notifies both C1 and C2 of the alteration. Conversely, if the user changes the size of a piece of the pie with the mouse, C1 reflects the modification to C who, in turn notifies C2.

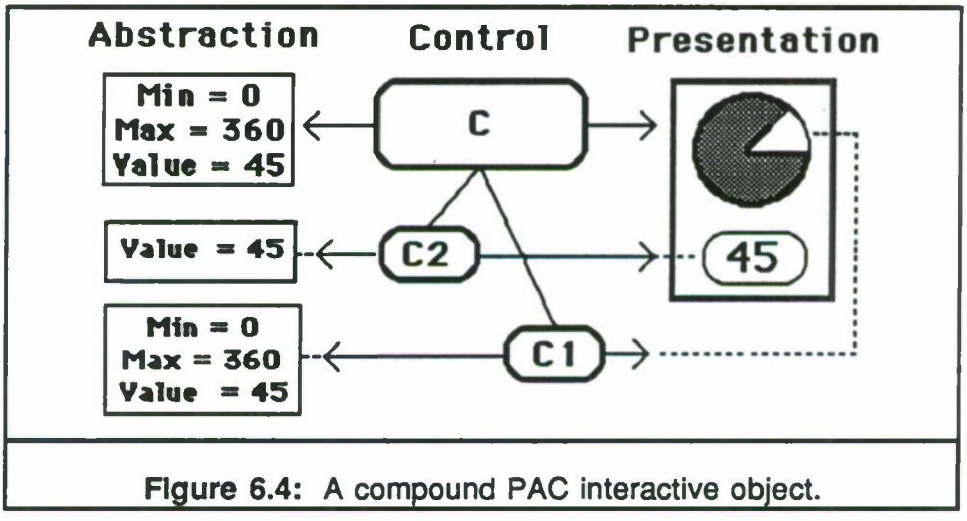


Figure 6.4: A compound PAC interactive object.

In summary, by applying PAC recursively at every level of abstraction of the user interface, everything in an interactive application is a PAC object, from the elementary Interactive object to the whole application. As shown in the upper rectangle of Figure 6.5, the whole interactive application is a PAC entity. The Abstraction part of the application involves three domain dependent concepts in the dialogue. The Controller at the top of the hierarchy bridges the gap between the Abstraction and the

Presentation. The Presentation is made of 4 interactive objects. The second lower rectangle shows the PAC structure of the compound interactive object represented as a black circle. This object is built from two elementary PAC objects and one compound object which, in turn, is composed of two elementary PAC objects.

In addition, the user interface of a workstation (generally refered to as a shell) may be modelled in a straighforward manner by adding an extra PAC layer on top of the application level. The Abstract part of that layer may include such global data structures as the "clipping board" or the "network status." The Presentation would present these data structures and allow for the initial invocation of applications. Finally, the Control part would, of course, bridge the gap between the abstract and the concrete sides. It would as well supervise the control parts of all of the active applications. Such an arbitrator should provide the basis for a uniform mechanism for transferring data between applications.

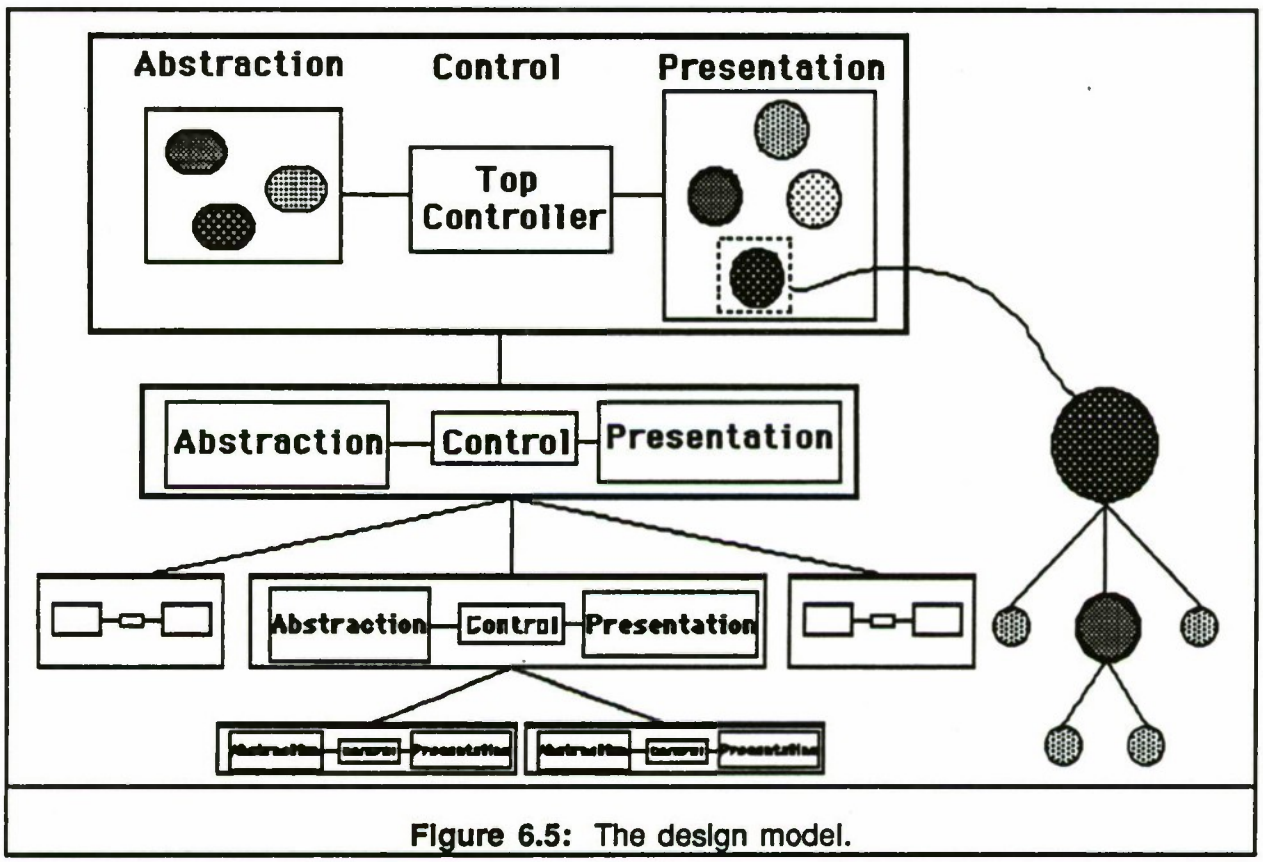


Figure 6.5: The design model.

This recursive object-oriented organization presents some advantages which are described in the following paragraph.

6.1.5.5. The Interest Aspects of the PAC Model

The PAC model has three interesting aspects:

1. It defines a consistent framework for the construction of user interfaces that is applicable at any level of abstraction. As a direct consequence, the units of exchange between the application (i.e., the Abstract part) and the UIMS (i.e., the PAC controller) are application concepts, not low-level details semantically irrelevant to the application.

2. It cleanly distinguishes functional notions from presentation policies and introduces the control part to bridge the gap between the abstract and the concrete worlds. The role of the control part may be extended from consistency maintenance between the two worlds, to the management of local contextual information that may be useful for help, error explanation and automatic adaptation to the user.

3. It takes full advantage of the object-oriented paradigm with the notion of interactive object.

An interactive object is an active entity. It evolves, communicates and maintains relationships with other objects. Such activity, parallelism and communication are automatically performed by the Object Machine, the generic class of the interactive objects. The Object Machine defines the general functioning that is made common to all of the interactive objects by means of the inheritance mechanism. In particular, each object owns a private finite state automaton for maintaining its current dialogue state. On receipt of a message, an object is thus able to determine which actions to undertake according to its current state. In particular, The PAC controller at the top of the hierarchy of controllers, maintains the global state of the dialogue with the application.

Interactive objects implement the dialogue in a distributed way. This feature can serve as a basis for the implemention of facilities related to the notion of context. It also provides the necessary grounds for concurrent multiple I/O in the following way. The set of automata (one automaton per interactive object) defines the global state of the interaction between the user and the application. The control of the interaction is therefore distributed in an evolutive network of interactive objects. Dialogue control is not handled by a unique monolithic dialogue manager difficult to maintain, extend and implement, in particular when one wants a pure user-driven style of interaction. Conversely, since interactive objects are able to maintain their own state, it is easy to let the user switch between objects in any order. Thus, an object-oriented approach provides for free the maintenance of the user's arbitrary manipulations.

Interactive objects are easily customizable. Object-oriented programming languages support data abstraction which makes it possible to change underlying implementations without changing the calling programs. In the present case, this principle allows the internal modification of an interactive object without changing its presentation and abstract interfaces. Interestingly, it also allows the modification of one interface without any side-effect on the other interface. For example, one can modify the presentation of an interactive object (such as attaching a different key translation table to an interactive object of type string) without reflecting on its abstract behaviour. This property makes possible fine grained dynamic adjustments of the user interface without massive modifications to the presentation of the whole application.

6.1.6. Multiple Views of Data

One problem associated with the separation of the user interface from the functional core of the application is the management of multiple presentations of the same application data item. Since the application is written to be media independent it has no knowledge of any presentation issues, in particular, how many times a particular piece of its data is presented to the user and in what forms.

For example, suppose the pressure within a pipe is represented both by the color of the fluid in the pipe and by a separate pressure gauge. When the pressure changes both presentations should change. Managing these multiple views of the same data item is the responsibility of the runtime kernel. The kernel must have a mechanism to determine which data items determine the nature of a particular presentation. Otherwise, the kernel cannot automatically manage the presentation. This mechanism must allow the determination that two different presentations depend upon the same data item.

The determination that two different presentations depend upon the same data item depends upon the interface between the functional core and the runtime kernel and the information presented to the runtime kernel. In Serpent, for example, two presentations are determined to depend upon the same data item if they both depend upon a particular element in the data base schema which describes the data. This allows the automatic modification of an aggregate in the presentation when a component changes if the runtime interface is in charge of maintaining the aggregate. It does not allow the automatic modification of the aggregate if the application is in charge of maintaining the aggregate.

6.1.7. Feedback

One of the most troublesome issues associated with the separation of the functional portion of the application from the user interface is that of feedback [Hudson 88].

Feedback is the displaying to the user some indication of the system's understanding of the actions being performed. For example, in the X toolkit, a widget will reverse video when the cursor is within the widget. It is possible to change cursor shape when the cursor goes from one window to another. These are examples of lexical feedback and are handled at the presentation level.

Another type of feedback comes from the runtime kernel. On the Macintosh, certain options within a menu are displayed in gray scale to indicate that they are not currently available. The runtime kernel knows the current context of the action and makes the decision to display certain items in a fashion that gives feedback to the end user about the current state of that item. This is an example of syntactic feedback (based on the current context).

A deeper level of feedback might be changing the color of a beam in a CAD/CAM application to represent the stress currently being placed on that beam. This is an example of semantic feedback since the determination of the current color depends upon knowledge that only the functional core of the application maintains.

These three types of feedback represent different levels of abstraction and should be performed in separate portions of the software. This implies that the software structure must be available to allow that separation. The hierarchical decomposition of PAC is explicitly designed to allow the separation of various levels of feedback.

The reason that feedback is a troubling issue is because of the performance implications. Feedback, by its nature, should be fast. The end user should, ideally, be given indications of the meaning of an action when that action is occurring. It is not clear that this is always possible in the case of deep semantic feedback and the architectural structure of a system may not always support both the performance requirements of rapid feedback and the separation of the functional core of the application from the user interface. In any case, the human processing model gives a bound on required functionality. Since events occurring in less than 0.1 second are seen to be instanteous, feedback performance requirements will be satisfied if they can be met within that time period.

6.2. User Interface Environments

6.2.1. Introduction

The actions of the runtime kernel are determined by a language used to describe the dialogue. The mechanism for specification of that language plays a large part in acceptability of the user interface runtime system. One possibility, which won't be further discussed, is to use a standard programming language to interact directly with the runtime kernel. MacApp [Schmucker 86], APEX [Coutaz 87a] and EZWin [Liebermann 85] are examples. The approach is to treat the runtime kernel as an extension of a toolkit.

More interesting are cases where specialized language or specification mechanisms exist. The examples to be discussed are:

1. Textual language specification
2. Graphical editor specification
3. Complete environments

Figure 6.6 represents the usage of the specification. The dialogue specifier creates a dialogue using some tool and the created specification provides the mechanism for the runtime kernel to operate. The specification can be distinct in time from the execution of the runtime kernel or specification time and runtime can be intertwined. The textual language specification which is discussed first is, inherently, distinct in time from runtime.

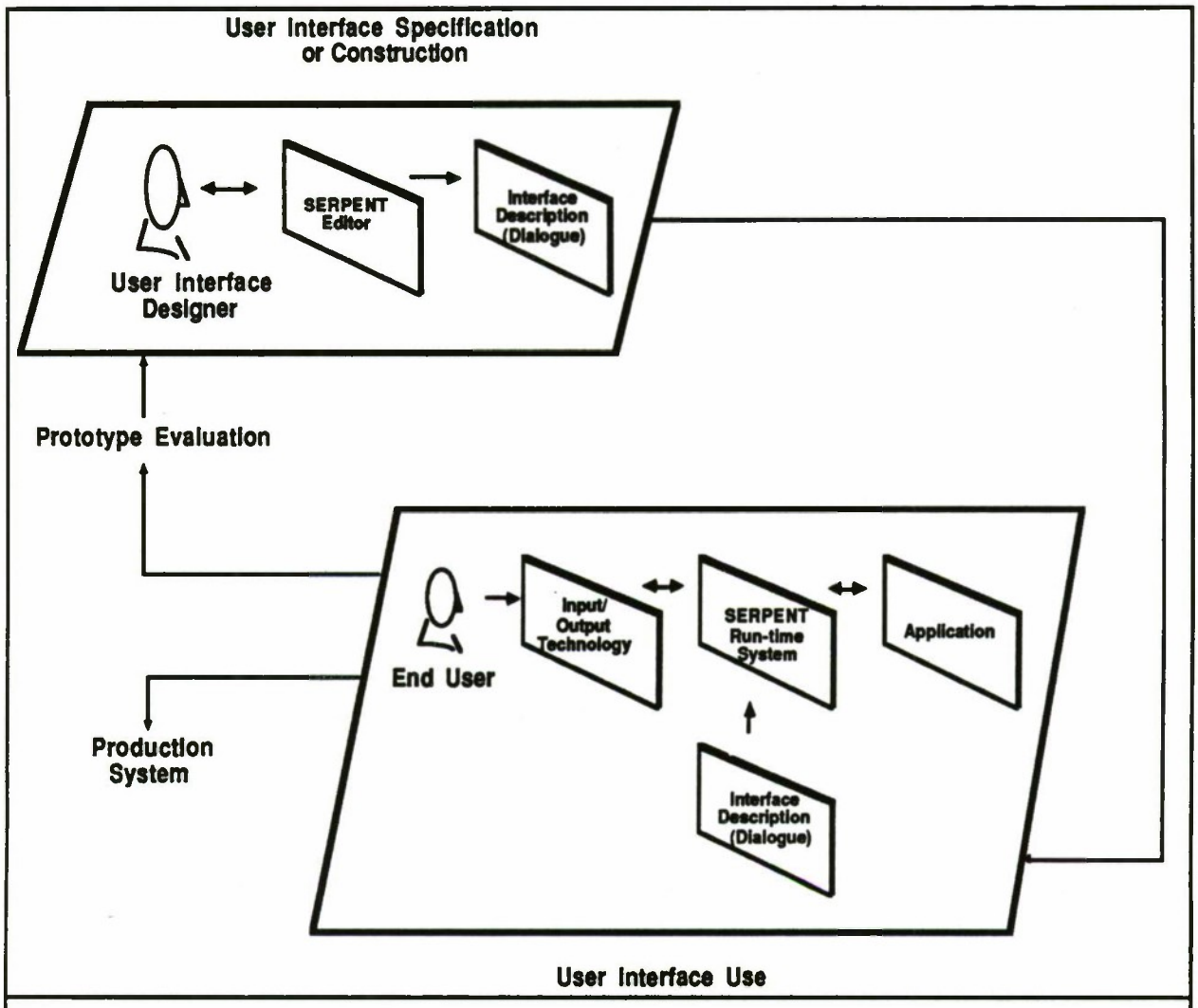


Figure 6.6: Specification of dialogues.

6.2.2. Textual language specification

Domain [Schulert 85] is a commercial user interface management system available from Apollo. The model that Domain uses is given in Figure 6.7. This is also the model used in Cousin [Hayes 83]. The interface between the domain dependent portion of the program and the user interface is defined to be a group of "Tasks". Each task has a computation portion. The user interface is defined in terms of building blocks which define the presentation in terms of the tasks. The application places values in the task which cause the presentation to change and the building blocks place values in the tasks which affect the application. Figure 6.8 shows the user interface for a simple example. Figure 6.9 gives the tasks, Figure 6.10 gives the building blocks and Figure 6.11 gives the application code for this example.

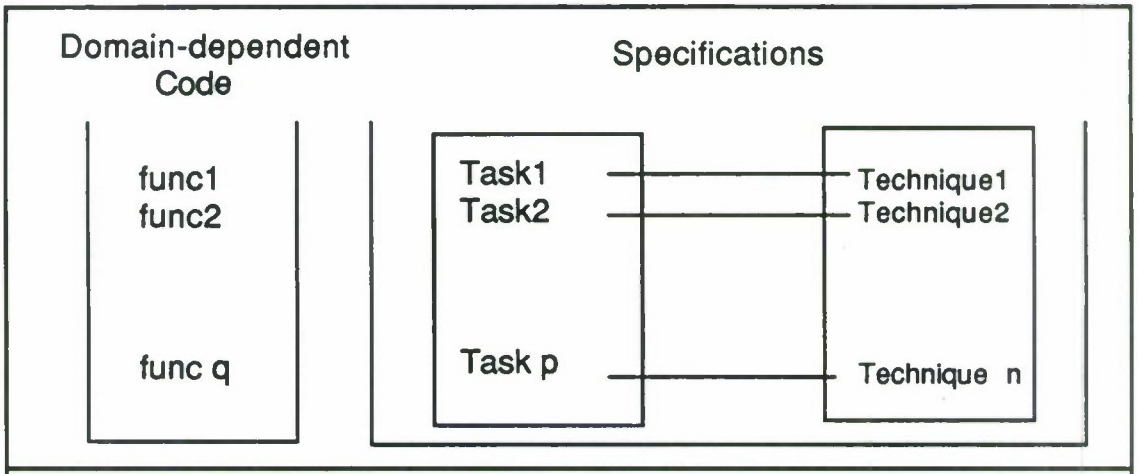


Figure 6.7: The model used by Domain.

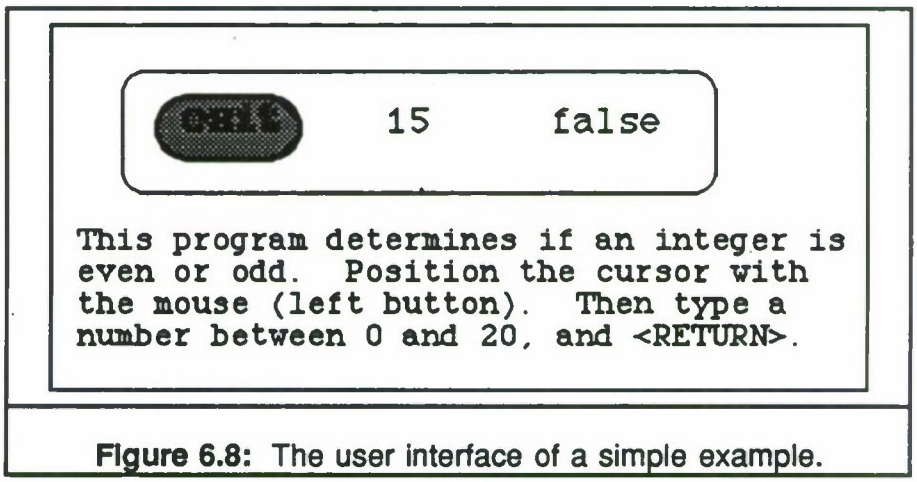


Figure 6.8: The user interface of a simple example.

APPLICATION-INTERFACE example

```
exit-task:=NULL:
    COMP => <CALL odd-or-even>
    MIN=0;
    MAX=20
    END

true-false-task:=BOOL:
    COMP => <>
    END

message-task:=MSG:
    VALUE =
            "This program determines if an integer is even or odd."
            &"Position the cursor with the mouse (left button)."
            &"Then type a number between 0 and 20, and <RETURN>."
    END
```

Figure 5.9: The tasks for the example of Figure 5.8.

```
USER-INTERFACE example

    exit:=ICON:
        TASK = exit-task;
        BACKGROUND = GREY;
        SHAPE = ROUNDED;
        SIZE = (100 350) PIXELS;
        STRING = "exit"
        END
    number:=INT_FIELD:
        TASK = number-task;
        BACKGROUND = OFF
        SHAPE = ROUNDED
        HELP-TEXT = "you must give an integer from 0 to 20"
        END
    true-false:=BOOL-FIELD
        TASK = true-false-task;
        BACKGROUND = OFF;
        SHAPE = ROUNDED;
        HELP -TEXT = "true=even number" & "false = odd number"
        END
    row-bottom:=ROW
        BACKGROUND = ON;
        ORIENTATION = HORIZONTAL;
        BORDER-WIDTH = 10; DIVISION -WIDTH = 5;
        OUTLINE = ON; SHAPE = ROUNDED;
        CONTENTS = (exit number true-false)
        END
    message:= DISPLAY TEXT
        TASK = message-task;
        SHAPE = ROUNDED
        END
row-all:= ROW
        BACKGROUND = ON;
        ORIENTATION = VERTICAL;
        BORDER-WIDTH = 10; DIVISION-WIDTH = 5;
        OUTLINE = ON; SHAPE = ROUNDED;
        CONTENTS = (row-bottom message)
        END
    std-window:
        CONTENTS = row-all
        END
```

Figure 6.10: The building blocks for the user interface for the example in Figure 6.8.

```
MAIN PROGRAM
    -- initiate DIALOG
    -- set initial values and defaults to tasks
        dp-$bool-set-value(true-false-task,true,status);
    -- activate a task or a group of tasks
        dp-$task-activate (dp-$all-task-group,...);
    -- wait for an input event
        dp-$event-wait (.....);
    -- exit Dialogue
        dp-$terminate (...);

A MODULE: the procedure which checks the parity
odd-or-even()
    int value-int, value-bool;
begin
    -- get input data
      dp-$int-get-value (number-task, value-int, status);
    -- check parity
      if ((value-int/2) == 0) then
          value-bool = true
      else
          value-bool = false;
    -- send the result to the task
      dp-$bool-set-value (true-false-task, value-bool, status);
end
```

Figure 6.11: The application code for the example of Figure 6.7.

6.2.3. Graphical Editor Specification

Since so much of the user interface is graphical in nature, it makes sense to have editors which are used to specify the graphical portion of the interface. Such editors have been created such as Menulay [Buxton 83]. The editors become layout editors. That is, the graphical editors are used to specify the appearance of a display and where on the display various presentation objects will reside. Once the layout has been specified then the connections between the presentation objects and the dialogue control are established. One problem with the usage of such editors is how to represent the dependencies upon application data. This issue goes to the heart of the timing distinction between specification time and runtime.

6.2.3.1. Realization

The dialogue gives a mapping between application objects and presentation objects. Implicit in this mapping is a dependency of certain attributes of the presentation object upon application values. If there were no such dependencies then the presentation would be totally independent of the application. When the display is presented to the specifier it must be realized with some set of application values. In order to be totally realistic, the values should be generated by the application and, hence, runtime and specification time are the same. In some systems (e.g. Serpent), the specifier provides fixed values for the attributes of the presentation objects which depend upon application objects. These fixed values then show the specifier one possible display. The problem of how to realize the interface leads into the idea of having a total environment for the development of user interface. Before discussing that issue,

however, some of the power possible with having a separate tool to construct the editor will be shown.

6.2.3.2. Smart Editors

When an interface is being constructed, typically there is a particular style being used for some of the components such as menus. Peridot [Myers 87] is a system that uses expert system techniques to make inferences about what style is being used for particular components. For example, the specifier would completely construct one menu and then whenever another menu was being constructed, Peridot would propose that it have the same style as the previous menu. This is one example of the type of intelligence that could be put into separate dialogue construction tools.

6.2.4. Environment

Although integrated user interface development and execution environments are desirable, they have not yet been produced. One system that comes close to an integrated environment is Hypercard [Harvey 88]. Hypercard is a system that manages textual and graphical objects in a multidimensional fashion. Each task that is to be accomplished is represented by a stack of cards. Cards within a stack can be linked to other stacks to represent associations that the specifier wishes to maintain. Cards can be searched to locate those that have information of relevance to the implementor.

Hypercard integrates the specification and the runtime by allowing scripts to be developed while data resides in the stacks. These scripts can then be executed, the results displayed and the scripts modified. This interaction between specification and execution allows the development of applications in a very smooth and continuous fashion.

Within Hypercard, the distinction between the application functional core and the user interface is blurred. This makes difficult the clear separation of functionality, which is the basis of the UIMS.

6.2.5. State of the Art

Within the field of user interfaces, today, we know how to do things which are application independent. Menus, scroll bars, etc are methods of allowing for user input with low-level feedback which have proven very valuable. What is not known is how to do things which are application dependent. Semantic feedback (feedback depending upon application semantics) is not well understood and current tools do a poor job of supporting this type of feedback while still providing a clear separation of functionality.

Bibliography

[Adobe 85] Adobe Systems, Incorporated.
Postcript Language Reference Manual.
Addison Wesley, 1985.

[Ahlers 86] K.L. Ahlers, A. Dwelly.
OUTILS: Towards a User Interface Management System for Graphical Interaction.
Technical Report ECRC (European Computer-Industry Research Centre), October 1986.

[Alletru 88] J.C. Alletru, F. Frédéric, O. Roussel, A. Vial.
IRENE, Système Expert d'Aide à la Configuration de Réseau XNS, Architecture Logicielle, Spécifications Externes.
Manuel Utilisateur; Rapport de Fin d'Etude, DESS-Génie Informatique.
Université Joseph Fourier, Juin 1988.

[Anderson 83] J.R. Anderson.
The Architecture of Cognition.
Harvard University Press, Cambridge, Massachusetts, 1983.

[Baecker 87] R.M. Baecker, W.A.S. Buxton.
Readings in Human-Computer Interaction: A Multidisciplinary Approach.
R.M. Baecker, W.A.S. Buxton (Editors).
Morgan Kaufmann Publishing, 1987.

[Barnard 81] P.J. Barnard, N.V. Hammond, J. Morton, J.B. Long, I.A. Clark.
Consistency and Compatibility in Human-Computer Dialogue.
International Journal of Man-Machine Studies, 15:87-134, 1981.

[Barnard 86] P.J. Barnard.
Cognitive Resources and the Learning of Human-Computer Dialogue.
MRC Applied Psychology Unit, 15 Chaucer Road, Cambridge, England, March 1986.

[Barnard 87] P.J. Barnard, M. Wilson, A. MacLean.
Approximate Modelling of Cognitive Activity: Towards an Expert System Design Aid.
In *Proceedings of the ACM CHI+GI Conference*, 21-26, April 1987.

[Barth 86] P. S. Barth.
An Object-Oriented Approach to Graphical Interfaces.
ACM Transactions on Graphics, 5(2), April 1986.

[Barthet 86] M. F. Barthet, C. Sibertin-Blanc.
La Modélisation d'Applications Interactives Adaptées aux Utilisateurs par des Réseaux de Petri à Structure de Donnée.
Actes du Troisième Colloque-Exposition de Génie Logiciel, Versailles, 117-136, Mai 1986.

[Bass 88] L. Bass, E. Hardy, K. Hoyt, R. Little, R. Seacord.
The Serpent Runtime Architecture and Dialogue Model.
Technical Report CMU/SEI-88-TR-6, ADA 196664, Carnegie Mellon University, Pittsburgh, PA 15213, January 1988.

[Blesser 82] T. Bleser, J.D. Foley.
Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces.
In *ACM Proceedings of Human Factors in Computer Systems Conference*, March 1982.

[Bly 86] S. Bly, J. K. Jarret.
A Comparison of Tiled and Overlapping Windows.
In *ACM Proceedings of the Computer Human Interaction Conference*, 101-106, 1986.

[Bobrow 83] D.G. Bobrow, M. Stefik.
The Loops Manual.
Technical Report KB-VLSI-81-13, Knowledge Systems Area, Xerox, Palo Alto Research Center, 1981.

[Borning 86a] A.H. Borning.
Graphically Defining New Building Blocks in ThingLab.
Human Computer Interaction, 2(4):269-295, 1986.

[Borning 86b] A.H. Borning.
Defining Constraints Graphically.
In *ACM Proceedings of the Computer Human Interaction Conference*, 137-143, 1986.

[Brownston 85] L. Brownston, R. Farrell, E. Kant, N. Martin.
Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.
Addison Wesley, 1985.

[Buxton 82] W. Buxton.
An Informal Study of Selection Positioning Tasks.
Graphics Interface'82, 323-328, 1982.

[Buxton 83a] W. Buxton.
Lexical and Pragmatic Considerations of Input Structures.
Computer Graphics, 31-37, January 1983.

[Buxton 83b] W. Buxton, M.R. Lamb, D. Sherman, K.C. Smith.
Towards a Comprehensive User Interface Management System.
Computer Graphics 17(3):35-42, July 1983.

[Buxton 86a] W. Buxton, B. Myers.
A Study in Two-Handed Input.
In *Proceedings of SIGCHI'86: Human Factors in Computing Systems Conference*, 321-326, 1986.

[Buxton 86b] W. Buxton.
There's More to Interaction Than Meets the Eye: Some Issues in Manual Input.
In Norman and Draper (Editors), *User Centered System Design: New Perspectives on Human-Computer Interaction*, 319-337.
Erlbaum, 1986.

[Cany 85] G. Cany.
Un Système d'Accès Poste de Travail sous UNIX.
Actes des Journées SM-90, 412-419.
Ed. Eyrolles, Décembre 1985.

[Card 83] S. Card, T. Moran, A. Newell.
The Psychology of Human-Computer Interaction.
ISBN 0-89859-243-7.
Lawrence Erlbaum Associates, 1983.

[Card 87] S. Card, A. Henderson.
A Multiple Virtual-Workspace Interface to Support User Task Switching.
In *Proceedings of the CHI+GI Conference on Human Factors in Computing Systems and Graphics Interface*, 1987.

[Cardelli 85] L. Cardelli, R. Pike.
Squeak: A Language for Communicating with Mice.
In *ACM Proceedings of Computer Graphics: SIGGRAPH'85 Conference*, 19(3):199-204, 1985.

[Cardelli 87] L. Cardelli.
Building User Interfaces by Direct Manipulation.
Digital Systems Research Center, Technical Report 22, October 1987.

[Carroll 83] J.M. Carroll.
Presentation and Form in User Interface Architecture.
Byte 8(12):113-122, December 1983.

[Carroll 84] J.M. Carroll, C. Carrithers.
Training Wheels in a User Interface.
Communication of the ACM, 27(8):800-807, August 1984.

[Carroll 85a] J.M. Carroll, D.S. Kay.
Prompting, Feedback and Error Correction in the Design of a Scenario Machine.
In *Proceedings of the CHI'85 Conference*, 149-153, April 1985.

[Carroll 85b] J.M. Carroll, R.L. Mack.
Metaphor, Computing Systems, and Active Learning. *International Journal of Man-Machine Studies*, 22(1):39-57, January 1985.

[Ciccarelli 84] E.C. Ciccarelli.
Presentation-Based User Interfaces.
Technical Report 794, Artificial Intelligence Laboratory, Massachusetts Intelligence Laboratory, August 1984.

[Cohen 86] E.S. Cohen, E.T. Smith, L.A. Iverson.
Constraint-Based Tiled Windows.
IEEE Computer Graphics and Applications, 6(5):35-45, May 1986.

[Conklin 87] J. Conklin.
Hypertext: An Introduction and Survey.
IEEE Computer, 20(9):17-41, September 1987.

[Coutaz 85a] J. Coutaz.
A Layout Abstraction for User System Design.
In *Proceedings of ACM SIGCHI*, 18-24, January 1985.
Also published as Carnegie Mellon University Technical Report, CMU-CS-84-167, December 1984.

[Coutaz 85b] J. Coutaz.
Abstractions for User Interface Design.
IEEE Computer, 18(9):21-34, September 1985.

[Coutaz 86a] J. Coutaz.
Abstractions for User Interface Toolkits.
In *Foundation for Human-Computer Communication*, 335-354, K. Hopper and I.A. Newman (Editors), North Holland,1986.

[Coutaz 86b] J. Coutaz.
Abstractions pour la Construction d'Interfaces Homme-Machine.
TSI, 5(4):239-250, Juillet-Août 1986.

[Coutaz 86c] J. Coutaz.
La Construction d'Interfaces Homme-Machine.
Rapport IMAG RR 635-I, Novembre 1986.

[Coutaz 87a] J. Coutaz, F. Berthier.
The Construction of User Interfaces.
In G. Bracchi and D. Tsichritzis (Editors), *Office Systems: Methods and Tools*, 59-66, North Holland, 1987.

[Coutaz 87a] J. Coutaz.
The Construction of User Interfaces and the Object-Oriented Paradigm.
In *Proceedings of the European Conference on Object-Oriented Programming*, 135-144, Paris, Juin 1987.

[Coutaz 87b] J. Coutaz.
PAC: An Implemention Model for Dialog Design.
Interact'87, 431-436, Stuttgart, Septembre 1987.

[Coutaz 87c] J. Coutaz.
PAC, an Object-Oriented Model for Implementing User Interfaces.
CHI+GI '87 Poster Session Papers, *ACM SIGCHI Bulletin*, 19(2):37-41, October 1987.

[Coutaz 88] J. Coutaz.
De L'Ergonome à L'Informaticien: Pour une Méthode de Conception et de Réalisation des Interfaces Homme-Machine.
Conférence invitée.
Actes du Colloque Européen ERGO-IA'88, Ergonomie et Intelligence Artificielle, Biarritz, Octobre 1988.

[diSessa 86] A.A. diSessa, H. Abelson.
Boxer: A Reconstructible Computational Medium. *Communications of the ACM*, 29(9):859-868, September 1986.

[Duce 87] D.A. Duce, F.R.A. Hopgood.
The Graphical Kernel System.
Computer-Aided Design, 19(8):396-409, October 1987.

[Duisberg 86] R.A. Duisberg.
Animated Graphical Interfaces Using Temporal Constraints.
In Proceedings of the ACM Computer Human Interaction Conference, 131-136, 1986.

[Ege 87] R.K. Ege, D. Maier, A. Borning.
The Filter Browser Defining Interfaces Graphically.
In *Proceedings of the European Conference on Object-Oriented Programming*, 155-165, Paris, Juin 1987.

[Foley 84] J. D. Foley, A. Van Dam.
Fundamentals of Interactive Computer Graphics.
Addison Wesley, 1984.

[Forgy 84] C.L. Forgy.
The OPS83 Report.
Technical Report CMU-CS-84-113, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1984.

[Garrett 82] M.T. Garrett, J.D. Foley.
Graphics Programming Using a Data Base System with Dependency Declarations.
ACM Transactions on Graphics, 1(2), April 1982.

[Garrett 86] L.N. Garrett, K.E. Smith.
Building a Timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application.
In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, 202-213, Portland, Oregon, September 1986.
Also in *Sigplan Notices*, 21(11):202-213, November 1986.

[Goldberg 84] A. Goldberg.
Smalltalk-80: The Interactive Programming Environment.
Addison-Wesley, 1984.

[Gosling 86a] J. Gosling.
Partitioning of Functions in Window Systems.
In Hopgood (editor), *Methodology of Window Management.*
Springer Verlag, 101-106, 1986.

[Gosling 86b] J. Gosling.
SunDew: A Distributed and Extensible Window System.
In *Proceedings of the Winter 1986 USENIX Conference*, 98-103, January 1986.

[Gosling 86c] J. Gosling, D. Rosenthal.
A Window Manager for BitMapped Displays and UNIX.
In Hopgood (Editor), *Methodology of Window Management.*
Springer Verlag, 101-106, 1986.

[Green 85] M.W. Green.
The Design of Graphical Interfaces.
Technical Report CSRI-170, Computer Systems Research Institute, University of Toronto, Canada, April 1985.

[Halasz 87] F.G. Halasz, T.P. Moran, R.H. Trigg.
NoteCards in a Nutshell.
In *Proceedings of ACM CHI+GI Conference*, 45-53,1987.

[Harel 87] D. Harel.
Statecharts: A Visual Formalism for Complex Systems.
Science of Computer Programming, 8(3):231-274, June 1987.

[Harvey 88] G. Harvey.
Understanding HyperCard for Version 1.1.
Sybex Book Publishers, 1988.

[Hayes 83] P.J. Hayes, P. Szekely.
Graceful Interaction Through the Cousin Command Interface.
International Journal of Man Machine Studies 19(3):285-305, September 1983.

[Hayes 85] P.J. Hayes, P. Szekely, R. Lerner.
Design Alternatives for User Interface Management Systems Based on Experience with Cousin.
In *Proceedings of the CHI'85 Conference*, 169-175, April 1985.

[Hayes-Roth 79] B. Hayes-Roth, F. Hayes-Roth.
A Cognitive Model for Planning.
Cognitive Science, 3:275-310, 1979.

[Henderson 86] A. Henderson.
The Trillium User Interface Design Environment.
In *Proceedings of SIGCHI'86: Human Factors in Computing Systems Conference*, 221-227, 1986.

[Hill 87a] R.D. Hill.
Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction.
In *ACM Transactions on Graphics* 5(2):179-210, April 1986.

[Hill 87b] R.D. Hill.
Event Response Systems: A Technique for Specifying Multi-Thread Dialogues.
In *Proceedings of the CHI+GI'87 Conference*, 241-248, 1987.

[Hudson 88] S.E. Hudson, R. King.
Semantic Feedback in the Higgens UIMS.
TR 88-14, Department of Computer Science, University of Arizona, Tucson, Arizona, March 1988.

[Hullot 86] J.M. Hullot.
SOS Interface: Un Générateur d'Interfaces Homme-Machine.
Actes des Journées Afcet-Informatique sur les Langages Orientés Objet, Bigre+Globule, 48, 69-78. Publ. IRISA, Campus de Beaulieu, 35042 Rennes, Janvier 1986.

[Hutchins 86] E. L. Hutchins, J. D. Hollan, D. A. Norman.
Direct Manipulation Interfaces: User-Centered System Design.
Lawrence Erlbaum Associates, 1986.

[ISO 85] International Organization for Standardization. Information Processing Systems - Computer Graphics - Graphical Kernel System (GKS) Functional Description. ISO IS 7942, July 1985.

[ISO 86a] International Organization for Standardization. Information Processing Systems - Computer Graphics - Programmer's Hierarchical Interface to Graphics (PHIGS) Functional Description. ISO DP 9592, October 1986.

[ISO 86b] International Organization for Standardization. Information Processing Systems - Computer Graphics - Techniques for Interfacing Graphical Devices (CGI) Functional Description. ISO DP 9636, December 1986.

[John85] B.E. John, P.S. Rosembloom. A Theory of Stimulus-Response Compatibility Applied to Human Computer Interaction. In *Proceedings of ACM CHI'85 Conference*, 213-220, 1985.

[John87] B.E. John, A. Newell. Predicting the Time Recall Computer Command Abbreviations. In *Proceedings of CHI+GI Conference*, 33-40, 1987.

[Karsenty 87] S. Karsenty. Graffiti: Un Outil Interactif et Graphique pour la Construction d'Interfaces Homme-Machine Adaptables. Thèse de Doctorat de 3ème Cycle Informatique, Université de Paris-Sud, Centre d'Orsay, Décembre 1987.

[Kasik 82] D. J. Kasik. A User Interface Management System. *Computer Graphics*, 99-106, July 1982.

[Kieras 85] D. Kieras, P.G. Polson. An Approach to the Formal Analysis of User Complexity. *International Journal of Man-Machine Studies*, 22:365-394, 1985.

[Kiger 84] J.I. Kiger. The Depth/Breadth Trade-Off in the Design of Menu-Driven User Interfaces. *International Journal of Man-Machine Studies*, 20:201-213, 1984.

[Knuth 79] D.E. Knuth. *TEX and Metafont: New Directions in Typesetting.* Digital Press, 1979.

[Lantz 84] K.A. Lantz, W.I. Nowicki.
Structured Graphics for Distributed Systems.
ACM Transactions on Graphics, 3(1), January 1984.

[Lantz 86] K.A. Lantz.
On User Interface Reference Models.
ACM SIGCHI Bulletin, 18(2):36-44, 1986.

[Lieberman 85] H. Lieberman.
There's More to Menu Systems Than Meets the Screen.
SIGGRAPH'85, Computer Graphics, 19(3):181-189, 1985.

[Lieberman 87] H. Lieberman.
Reversible Object-Oriented Interpreters.
In *Proceedings of the European Conference on Object-Oriented Programming*, 13-22, Paris, Juin 1987.

[Lindsay 80] P.H. Lindsay, D.A. Norman.
Human Information Processing: An Introduction to Psychology, second edition.
Edition Etudes Vivantes 6700 Chemin Côte de Liesse, Saint-Laurent, Quebec, 1980.

[Linton 86] M. Linton, C. Dunwoody.
Partitioning User Interfaces with Interactive Views.
Computer Systems Laboratory, Stanford University, Stanford, California 94305-2192, Private Communication, April 1986.

[Lunati 88] J.M. Lunati, V. Normand.
Un Serveur d'Interaction Centré Objet.
Projet de Fin d'Etudes, Institut National Polytechnique de Grenoble E.N.S.M.A.G. Juin 1988.

[Lynch 86] G. Lynch, J. Meads.
In Search of a User Interface Reference Model: Report on the SIGCHI Workshop on User Interface Reference Models. *SIGCHI Bulletin*, 18(2):25-33, October 1986.

[Meyer 87] B. Meyer.
Reusability: The Case for Object-Oriented Design.
IEEE Software, 50-59, March 1987.

[Meyrowitz 86] N. Meyrowitz.
Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework.
In *Proceedings of the Object-Oriented Programming Systems Languages and Applications Conference*, September 1986.
Also in *Sigplan Notices*, 21(11):186-201, November 1986.

[Michard 82] A. Michard.
Graphical Presentation of Boolean Expressions in a Data Base Query Language: Design Notes and an Ergonomic Evaluation.
Behaviour and Information Technology, 1(3):279-288, 1982.

[Mikelsons 81] M. Mikelsons.
Prettyprinting in an Interactive Programming Environment.
In *Proceedings of the ACM Sigplan SIGOA Symposium on Text Manipulation*, June 1981.

[Miller 81] D.P. Miller.
The Depth/Breadth Trade-Off in Hierarchical Computer Menus.
In *Proceedings of the 25th Annual Meeting of the Human Factors Society*, 296-300, 1981.

[Miller 75] G.A. Miller.
The Psychology of Communication, second edition.
Basic Books, New York, 1975.

[Moran 81] T. Moran.
The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems.
International Journal of Man-Machine Studies, 15:3-50, 1981.

[Moran 83] T.P. Moran.
Getting into a System: External-Internal Task Mapping Analysis.
In *Proceedings of the Computer Human Interaction'83 Conference*, 1983.
Also in *ACM SIGCHI Bulletin* special issue, 45-49, 1983.

[Morcos 86] E. Morcos-Chounet, M.J. Brossard, A. Conchon.
Affichage Interactif d'Arbres Abstraits.
Troisième Colloque-Exposition de Génie Logiciel, 137-150, Versailles, AFCET, Mai 1986.

[Myers 84] B.A. Myers.
The User Interface of Sapphire.
IEEE Computer Graphics and Applications 4(12):13-23, December 1984.

[Myers 86] B.A. Myers.
Visual Programming, Programming by Example and Program Visualization: A Taxonomy.
In *Proceedings of CHI'86 Conference, Human Factors in Computing Systems*, 59-66, 1986.

[Myers 87] B.A. Myers.
Creating Dynamic Interaction Techniques by Demonstration.
In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, 271-278, 1987.

[Nanard 84] J. Nanard, M. Nanard.
Manipulation Interactive de Documents.
Techniques et Science Informatiques, 3(6):443-451, 1984.

[Neal 87] L. R. Neal.
Cognition-Sensitive Design and User Modelling for Syntax-Directed Editors.
In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, 99-102, 1987.

[Nelson 85] G. Nelson.
Juno: A Constraint-Based Graphics System.
In *Proceedings of Computer Graphics: SIGGRAPH'85 Conference*, 19(3):235-243, July 1985.

[Newell 86] A. Newell, S. Card.
Straightening Out Softening Up: Response to Carroll and Campbell.
Human Computer Interaction, 2(3):251-267, 1986.

[Nielsen 87] J. Nielsen.
The Spectrum of Models in Software Ergonomics.
In *Proceedings of the Fifth Symposium on Empirical Foundations of Information and Software Sciences*, 1987.

[Nievergelt 80] J. Nievergelt, J. Weydert.
Sites, Modes, and Trails: Telling the User of an Interactive System Where He Is, What He Can Do, and How to Get Places.
In R.A. Guedj, P. Ten Hagen, F.R. Hopgood, H. Tucker, D.A. Duce (Editors), *Dans Methodology of Interaction*.
North Holland, 327-338, 1980.

[Norman 86] D. A. Norman, S. W. Draper.
User-Centered System Design.
Lawrence Erlbaum Associates, 1986.

[Olsen 83] D.R Olsen, E.P Dempsey.
Syngraph: A Graphical User Interface Generator.
Computer Graphics, 43-50, July 1983.

[Pfaff 85] User Interface Management Systems.
In G. E. Pfaff (Editor), *Eurographics Seminars*,
Springer-Verlag, 1985.

[Polson 85] P.G. Polson, D.E. Kieras.
A Quantitative Model of the Learning and Performance of Text Editing Knowledge.
In *Proceedings of the ACM CHI'85 Conference*, 207-212, 1985.

[Price 83] L.A. Price, C.A. Cordova.
Use of Mouse Buttons.
In *Proceedings of the ACM CHI'83 Conference*, 263-266, 1983.

[Quint 87] V. Quint.
Une Approche de l'Edition Structurée des Documents.
Thèse de Doctorat d'Etat, Université Scientifique, Technologique et Médicale de Grenoble, 1987.

[Rose 86] C. Rose et al.
Inside Macintosh.
Addison Wesley, 1986.

[Rosson 83] M.B. Rosson.
Patterns of Experience in Text Editing.
In *Proceedings of CHI'83 Conference on Human Factors in Computer Systems*, 177-183, 1983.

[Sacerdoti 74] E.D. Sacerdoti.
Planning in a Hierarchy of Abstraction Spaces.
Artificial Intelligence, 5:115-135, 1974.

[Scapin 87] D.L. Scapin.
Guide Ergonomique de Conception des Interfaces Homme-Machine.
Rapport INRIA 77, Octobre 1987.

[Scheifler 86] R.W. Scheifler, J. Getty.
The X Window System.
ACM Transactions on Graphics 5(2):79-109, April 1986.

[Schmucker 86] K. Schmucker.
MacApp: An Application Framework.
Byte 11(8):189-193, 1986.

[Schulert 85] A.J. Schulert, G.T. Rogers, J.A. Hamilton.
ADM - A Dialog Manager.
In *Proceedings of the CHI'85 Conference*, 177-183, April 1985.

[Shneiderman 87] B. Shneiderman.
Designing the User Interface: Strategies for Effective Human-Computer Interaction.
Addison Wesley, 1987.

[Shuey 86] D. Shuey, D. Bailey, T.P. Morrissey.
PHIGS: A Standard, Dynamic, Interactive Graphics Interface.
IEEE Computer Graphics and Application, 50-57, August 1986.

[Shuey 87] D. Shuey.
PHIGS: A Graphics Platform for CAD Application Development.
Computer-Aided Design, 19(8):410-417, October 1987.

[Sibert 86] J.L. Sibert, W.D. Hurley, T.W. Bleser.
An Object-Oriented User Interface Management System.
SIGGRAPH'86, 20(4):259-268, 1986.

[Simon 84] H. A. Simon.
The Sciences of the Artificial, third edition.
The MIT Press, 1984.

[Sisson 86] N. Sisson.
Dialogue Management Reference Model.
ACM SIGCHI, 18(2):34-35, October 1986.

[Smith 82] D.C. Smith, C. Irby, R. Kimball, B. Verplank.
Designing the Star User Interface.
Byte, 7(4):242-282, April 1982.

[Smith 87] R.B. Smith.
Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic.
In *Proceedings of the ACM Computer Human Interaction Conference*, 61-67, 1987.

[Stefik 87] M. Stefik, G. Foster, D.G. Bobrow, K. Hahn, S. Lanning, L. Suchman.
Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings.
Communications of the ACM, 30(1):32-47, January 1987.

[SUN 87] SUN Microsystems, Inc.
NeWS Manual.
SUN Microsystems, Inc., 2250 Garcia Avenue, Mountain View, California 94043.

[Tanner 83] P. Tanner, W. Buxton.
Some Issues in Future User Interface Management Systems (UIMS) Development.
In *Proceedings of IFIP Working Group 5.2 Workshop on User Interface Management*, November 1983.

[Tanner 86] P. Tanner, S.A. Mackay, D. A. Stewart.
A Multitasking Switchboard Approach to User Interface Management.
Computer Graphics: SIGGRAPH86 Conference Proceedings, 20(4):241-248, 1986.

[Trigg 87] R.H. Trigg, T.P. Moran, F.G. Halasz.
Adaptability and Tailorability in NoteCards.
In H.J. Bullinger, B.Shackel (Editors), *Conference Proceedings of Human-Computer Interaction Interact'87*, 723-728, North Holland, 1987.

[Tufte 83] E.R. Tufte.
The Visual Display of Quantitative Information.
Graphics Press, Box 430, Cheshire, Connecticut 06410, 1983.

[Tullis 85] T.S. Tullis.
Designing a Menu-Based Interface to an Operating System.
In *Proceedings of CHI'85 Conference*, 79-84, 1985.

[Uebbing 86] J. Uebbing, C. Young.
User Interface Performance Issues.
Byte 11(8):176-176, 1986.

[Warnock 82] J. Warnock, D.K. Wyatt.
A Device-Independent Graphics Imaging Model for Use with Raster Devices.
Computer Graphics, 16(3):313-319, July 1982.

[Wasserman 85] A. Wasserman.
Extending State Transition Diagrams for the Specification of Human-Computer Interaction.
IEEE Transactions on Software Engineering, 11(8), August 1985.

[Wong 82] P.C.S. Wong, E.R. Reid.
Flair: User Interface Design Tool.
ACM Computer Graphics, 16(3):87-98, July 1982.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-TR-4	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-89-004
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.	6b. OFFICE SYMBOL (If applicable) SEI
	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE	8b. OFFICE SYMBOL (If applicable) ESD/XRS1
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) HUMAN-MACHINE INTERACTION CONSIDERATIONS FOR INTERACTIVE SOFTWARE	
12. PERSONAL AUTHOR(S) Len Bass, Joelle Coutaz	
13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM _____ TO _____
14. DATE OF REPORT (Yr., Mo., Day) February 1989	15. PAGE COUNT 112
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	cognitive model user interface windowing systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document introduces current concepts and techniques relevant to the design and implementation of user interfaces. A user interface refers to those aspects of a system that the user refers to, preceives, knows and understands. A user interface is implemented by code that mediates between a user and a syste. This document covers both aspects.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE